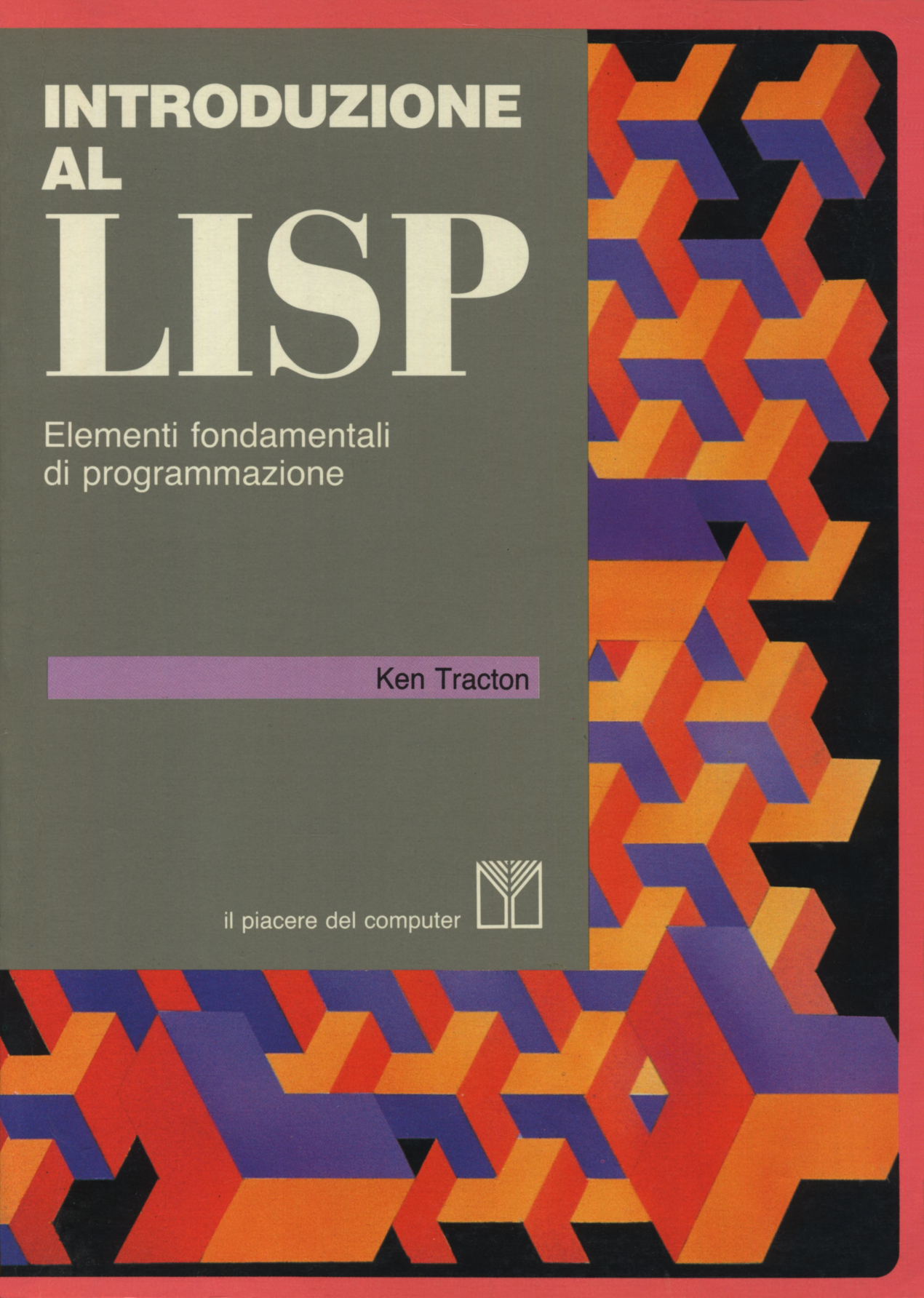
INTRODUZIONE AL LISP

Elementi fondamentali di programmazione

Ken Tracton

il piacere del computer

Il piacere del computer

1 *Tom Rugg e Phil Feldman* 32 programmi con il PET
2 *Rich Didday* Intervista sul personal computer, hardware
3 *Tom Rugg e Phil Feldman* 32 programmi con l'Apple
4 *Ken Knecht* Microsoft Basic
5 *Paul M. Chirlian* Pascal
6 *Tom Rugg e Phil Feldman* 32 programmi con il TRS-80
7 *Rich Didday* Intervista sul personal computer, software
8 *Herbert D. Peckham* Imparate il Basic con il PET/CBM
9 *Karl Townsend e Merl Miller* Il personal computer come professione
10 *Karen Billings e David Moursund* Te ne intendi di computer?
11 *Thomas Dwyer e Margot Critchfield* Il Basic e il personal computer, uno: introduzione
12 *Don Inman e Kurt Inman* Imparate il linguaggio dell'Apple
13 *Thomas Dwyer e Margot Critchfield* Il Basic e il personal computer, due: applicazioni
14 *Luigi Pierro* Il manuale del CP/M
15 *Carlo Sintini* A scuola con il PET/CBM
16 *David Johnson-Davies* Il manuale dell'Atom
17 *David E. Schultz* Il libro del Commodore VIC 20
18 *Jim Huffman e Robert Bruce* Il "debug" nei personal computer
19 *John M. Nevison* Programmazione in Basic per l'uomo d'affari
20 *Mark Harrison* Imparate il Basic con lo ZX81
21 *Ronald W. Anderson* Dal Basic al Pascal
22 *Herbert D. Peckham* Imparate il Basic con il Texas TI 99/44
23 *Sergio Borsani* A scuola con il Texas TI 99/4A
24 *Jerry Willis e Deborrah Willis* Come usare il Commodore 64
25 *Mark Harrison* Imparate il Basic con lo Spectrum
26 *Carlo Sintini e Costantino Mustacchio* A scuola con il Commodore 64
27 *David A. Lien* Imparate il Basic con l'IBM PC
28 *Ken Tracton* Introduzione al Lisp
29 *Fabio Mavaracchio* Programmi in Basic per l'elettronica

Ken Tracton

Introduzione al Lisp

franco muzzio & c. editore

Indice generale

Prefazione

Se avete a disposizione un computer con il linguaggio Lisp, potete cominciare a usare il Lisp mentre leggete questo libro: il modo migliore per imparare qualunque linguaggio è quello di cominciare subito a usarlo.
La prima sezione del libro procede a domande e risposte, con paragrafi di "ripasso" e moltissimi esempi. La seconda sezione offre invece numerosi programmi e routine in Lisp, di varia natura.
Il lettore che desidera effettivamente apprendere, se seguirà tutte le domande e risposte, otterrà tutte le informazioni necessarie per capire qualunque programma in Lisp.
Verso la fine della prima sezione troverete l'affermazione che sistemi Lisp diversi possono trattare talune cose in forma leggermente diversa. Per esempio, scorrendo i programmi non sorprendetevi al vedere in taluni punti i simboli " ' ": hanno esattamente lo stesso significato della parola "quote" descritta nel testo. L'interprete Lisp usato durante la preparazione del libro aveva questo simbolo invece della più comune funzione quote. Tutti i programmi e le definizioni sono stati controllati su un computer CDC CYBER.

Ken Tracton

CAPITOLO UNO

Gli elementi fondamentali del Lisp

Il Lisp, probabilmente il più noto fra i linguaggi dell'intelligenza artificiale, fu inventato da John McCarthy e in genere è considerato un linguaggio per la descrizione di processi. Viene utilizzato nel campo dell'intelligenza artificiale perché è preciso, privo di ambiguità e relativamente facile da apprendere: la stessa idea di fondo dell'intelligenza artificiale sarebbe perduta, se non fosse per il Lisp e gli altri linguaggi della stessa famiglia.

Gli enunciati (*statement*) in Lisp (fatta eccezione per il cosiddetto livello 0) sono sempre racchiusi fra parentesi. Una *lista* è una coppia di parentesi con ciò che contengono; gli elementi contenuti sono detti elementi della lista. Il Lisp è un linguaggio di tipo "prefisso": le funzioni precedono gli argomenti. È un po' simile alla notazione RPN dei calcolatori tascabili, ma a rovescio.

Delle molte funzioni disponibili nel Lisp, la maggior parte sono mnemoniche. È bene fare attenzione a studiarle in ordine, per evitare confusione. Gli elementi in Lisp in genere sono chiamati *atomi*; liste e atomi costituiscono le *S-espressioni* (espressioni simboliche).

Vi sono funzioni matematiche che usano parole anziché simboli. Alcune funzioni costruiscono liste, altre scompongono liste nei loro componenti. Vi sono modi per inibire la valutazione di una funzione e modi per provocare la valutazione di una funzione: valutazione è il termine che si riserva per l'esecuzione di una funzione. Vi sono anche funzioni speciali, chiamate predicati, che sono come i predicati nel linguaggio ordinario.

Queste funzioni hanno come valore o il vero o il falso. Di solito la parola "falso" è sostituita dalla parola NIL.

FUNZIONI ALGEBRICHE

In Basic il logaritmo di X si indica LOG(X)?
Sì, in Basic tutte le funzioni sono scritte in questo modo: il nome della "funzione" seguito fra parentesi dall'argomento della funzione (fig. 1.1).

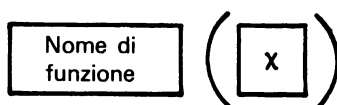



Fig. 1.1.

In Lisp il logaritmo di X si scrive LOG(X)?
No, in Lisp tutte le funzioni sono scritte nella forma ("nome della funzione" "variabile") (fig. 1.2). Pertanto il logaritmo di X viene indicato come: (LOG X).

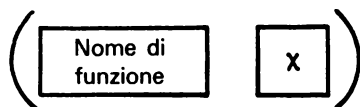



Fig. 1.2.

Questo si applica a tutte le funzioni matematiche nel Lisp?
Sì, tutte le funzioni sono racchiuse fra parentesi.

Come si scrive la somma di X e Y in Lisp (fig. 1.3)?
(PLUS X Y).

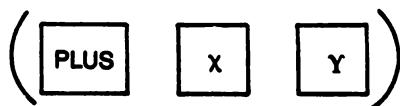



Fig. 1.3.

Rispetto alle manipolazioni algebriche, in che cosa il Lisp differisce dal Basic?
Basic, Fortran e APL usano un linguaggio algebrico. Le funzioni comuni come somma, differenza, divisione e prodotto vengono trattate in un modo particolare perché risulti più veloce la scrittura del programma. In questi linguaggi algebrici, le funzioni algebriche primitive sono scritte

senza parentesi. Il Lisp, invece, è un linguaggio "funzionale". Tutte le funzioni, primitive o meno, sono scritte fra parentesi.

Come si indica il logaritmo del seno di X in Lisp?
Si scrive con le parentesi (fig. 1.4):

(LOG(SIN X)).

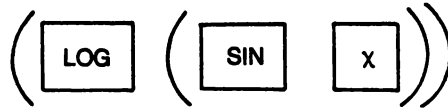



Fig. 1.4.

Come si scriverebbe in Lisp X/Z-Y?
Questa espressione viene scritta così (fig. 1.5):

(QUOTIENT X (DIFFERENCE Z Y)).

*Come si scrive X - Y*Z in Lisp?*
Così:

(DIFFERENCE X (TIMES Y Z)).

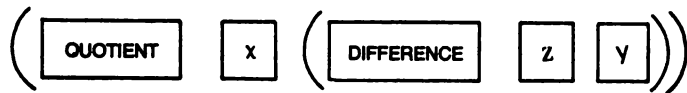



Fig. 1.5.

Si può scrivere qualunque espressione aritmetica in Lisp, in questo modo?
Sì, qualunque funzione di natura aritmetica, comprese espressioni che facciano uso di parentesi, può essere scritta in questo modo.

*Come si scriverebbe (A-B) * (Z-X)?*
Questa espressione va scritta (fig. 1.6):

(TIMES(DIFFERENCE A B) (DIFFERENCE Z X).

Fig. 1.6

*In Lisp si può scrivere A * (-B)?*

Sì, in questo modo (fig. 1.7):

(TIMES A (MINUS B)).

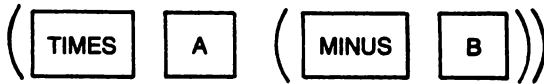



Fig. 1.7.

Qual è la differenza fra DIFFERENCE *e* MINUS*?*
La funzione DIFFERENCE corrisponde alla sottrazione ordinaria, mentre la funzione "meno unaria" è scritta (fig. 1.8) come:

(MINUS "variabile").

Pertanto, per indicare un numero negativo, in Lisp bisogna scrivere (MINUS X) e mai (-X).

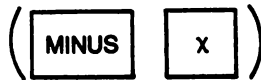



Fig. 1.8.

In Lisp le funzioni algebriche hanno sempre due argomenti?
Sì, in Lisp tutte le funzioni algebriche hanno almeno due argomenti, fatta eccezione per la funzione MINUS, che è unaria.

Come si scrive la funzione che dà la somma di tre o più grandezze?
Si scrive (fig. 1.9):

(PLUS A B C)

che ha lo stesso significato di:

(PLUS A(PLUS B C)).

ha lo stesso significato di

Fig. 1.9.

Questo procedimento si applica anche alla moltiplicazione?
Sì, si può scrivere X * Y * Z nella forma (fig. 1.10):

(TIMES X Y Z),

invece di scrivere

(TIMES X (TIMES Y Z)).

invece di scrivere

Fig. 1.10.

Quali sono le funzioni aritmetiche in Lisp?
Sono PLUS (addizione), DIFFERENCE (sottrazione), TIMES (moltiplicazione), QUOTIENT (divisione) e REMAINDER (il resto, il valore che rimane dopo una divisione).

Che cos'è esattamente la funzione REMAINDER*?*
La funzione REMAINDER applicata a X e Y dà come valore il resto della divisione di X per Y. Per esempio (fig. 1.11), se scriviamo:

(REMAINDER X Y)

e X e Y sono rispettivamente 12 e 5, il valore della funzione, cioè il resto, sarebbe 2.

Fig. 1.11.

Sono disponibili altre funzioni matematiche nel Lisp?
Sì: ne parleremo più avanti, però.

ESPRESSIONI

L'uso delle funzioni in Lisp ha vantaggi o svantaggi?
Usando le funzioni in questo modo si hanno sia vantaggi che svantaggi.

Quali sono gli svantaggi?
Gli svantaggi principali del metodo "funzionale" del Lisp sono le espressioni più lunghe e la necessità di un elevato numero di parentesi (fig. 1.12). Per esempio, è sicuramente più facile scrivere in Basic:

SIN (X * X - 4)*A/B

che non scrivere in Lisp:

(TIMES(SIN(DIFFERENCE(TIMES X X)4)) (QUOTIENT A B)).

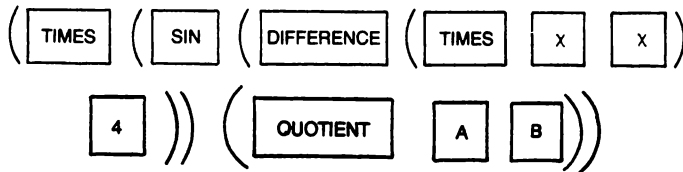



Fig. 1.12.

E i vantaggi quali sono?
Il vantaggio della notazione funzionale del Lisp è la sua capacità di unificare il linguaggio. In Lisp, questo è l'unico tipo di costruzione disponibile. Ogni caratteristica del Lisp, per esempio il trasferimento di controllo, i test condizionali, le definizioni ecc., viene "definita" creando una funzione particolare in grado di gestirla.

È sempre necessario usare tutte queste parentesi per la semplice aritmetica?
Per il Lisp standard, sì; tuttavia esiste un particolare dialetto del Lisp, chiamato MLisp, studiato in modo da realizzare le funzioni aritmetiche senza dover fare ricorso a "tutte" queste parentesi.

Come si chiamano le espressioni usate fin qui?
Sono chiamate S-espressioni.

Quanti tipi di componenti si trovano in una S-espressione?
Due: gli "elementi" come le cifre (0 1 2 ... 9) e i caratteri alfabetici (A B C ... Z) da una parte e le parentesi dall'altra (fig. 1.13).

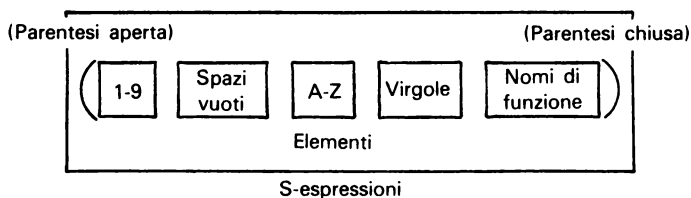



Fig. 1.13.

Come vengono chiamati in Lisp le cifre e i simboli alfabetici?
Sono chiamati atomi.

Qualunque espressione che contenga atomi e parentesi è una S-espressione?
No, perché sia una S-espressione devono essere seguite determinate regole di costruzione.

Queste sono S-espressioni?

```
(TIMES V N))
)PLUS 4 5)
(DIFFERENCE A B)
```

Le prime due no: la prima perché termina con due parentesi, la seconda perché inizia con una parentesi chiusa. La terza invece è una S-espressione.

Come si definisce una S-espressione?
Una S-espressione è costituita da atomi e parentesi. Deve esistere un equilibrio fra il numero delle parentesi aperte e il numero delle parentesi chiuse. *Non deve* esserci inversione di parentesi e le parentesi *non devono* essere nel posto sbagliato.

C'è qualche altro aspetto importante delle S-espressioni?
Sì, bisogna considerare anche gli spazi vuoti. Nel Basic e nel Fortran gli spazi vuoti sono ignorati, mentre in Lisp sono molto importanti per l'esecuzione delle funzioni.

Per che cosa sono usati gli spazi vuoti?
Per separare gli argomenti.

Si può usare qualche altro simbolo, al posto degli spazi?
Si possono usare le virgole, ma i programmatori in genere preferiscono utilizzare gli spazi vuoti.

In che modo le virgole possono sostituire gli spazi?
Una virgola può essere collocata in qualunque punto in cui avrebbe dovuto esserci uno spazio (fig. 1.14). Chiariranno meglio l'idea questi esempi:

```
(TIMES C B) ........(TIMES,C B)
                    (TIMES C,B)
                    (TIMES,C,B)
```

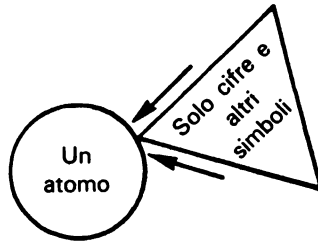



Fig. 1.14. Come nella fisica classica, in Lisp un atomo non può essere diviso in elementi più semplici.

Si può usare più di uno spazio in un punto dato e c'è qualche caso in cui gli spazi possono essere tralasciati?
Nella maggior parte dei linguaggi di programmazione è possibile inserire una stringa di spazi dovunque sia richiesto almeno uno spazio. Inoltre è possibile tralasciare lo spazio prima o dopo una parentesi aperta o chiusa.

Queste due S-espressioni sono lecite in Lisp?

```
(TIMES (SIN (DIFFERENCE (TIMES X X ) 4 )) (QUOTIENT A B ))
(TIMES (PLUS X C) H (PLUS A B) (PLUS R T))
```

Sì, sono ambedue lecite e accettabili in Lisp. È stato solo aumentato il numero degli spazi. La S-espressione che segue è lecita perché sono stati eliminati solo gli spazi che precedono le parentesi aperte e chiuse:

```
(TIMES(PLUS X C) H(PLUS A B)(PLUS R T))
```

Queste regole possono essere applicate a qualunque funzione in Lisp?
Sì, la regola si applica a qualunque costruzione in Lisp.

ATOMI

Come si chiamano le costanti e le variabili in una S-espressione?
Si chiamano atomi.

Quali regole seguono le variabili in Lisp?
Le variabili seguono la regola comune per gli identificatori. *Devono* essere composte da lettere o numeri e *devono* cominciare con una lettera.

In che cosa differiscono le variabili in Lisp da quelle degli altri linguaggi?

La differenza principale sta nel maggior numero di caratteri permessi nel nome della variabile. La maggior parte dei linguaggi consente l'uso solo di un certo numero di caratteri, mentre in Lisp per la costruzione di un nome di variabile non c'è alcun limite al numero dei caratteri (fig. 1.15).

Altri
linguaggi:
nome di variabile breve

Lisp Questo è un nome di variabile lungo

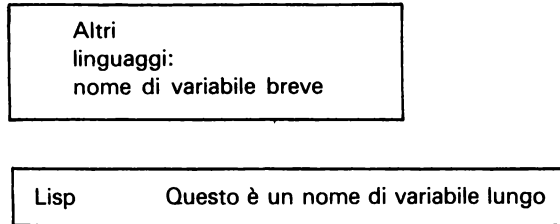



Fig. 1.15.

Esistono in Lisp identificatori particolari?
Sì, T, F e NIL.

Che cosa succede se si usa uno di questi identificatori particolari (riservati)?
Tipicamente, il risultato non è prevedibile.

Per che cosa sono usate queste variabili riservate?
T e F significano rispettivamente "vero" (*true* in inglese) e "falso" (*false*); NIL ha vari usi.

Quali sono gli usi di NIL *nel Lisp?*
Di molti parleremo più avanti, ma val la pena di citarne almeno uno subito. Molto spesso in Lisp NIL è usato per sostituire F. In effetti, per indicare "falso" viene usato più spesso NIL che non F.

Che cos'è un atomo?
Un atomo è una qualunque stringa di caratteri (fig. 1.16.).

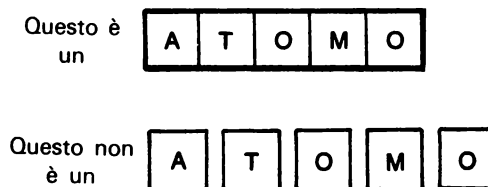



Fig. 1.16.

Una variabile in Lisp può avere un valore?
Sì, in Lisp, come in Fortran o in Basic, una variabile può avere un valore.

Qual è un modo per assegnare un valore a una variabile in Lisp?
Si può usare la funzione SETQ.

Come opera la funzione SETQ*?*
Questa è una funzione SETQ valida:

(SET Q Z 10)

Questa S-espressione pone il valore di Z uguale a 10. Il primo argomento è la variabile, mentre il secondo è il valore che le si vuole assegnare (fig. 1.17).

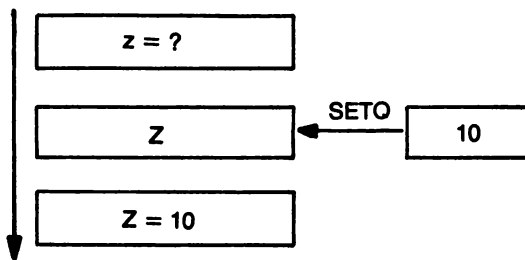



Fig. 1.17.

Il secondo argomento può essere qualcosa di più complesso di una semplice quantità numerica?
Il secondo argomento può avere tutta la complessità che si vuole. Queste sono espressioni SETQ valide:

```
(SETQ X Y)
(SETQ X (PLUS A B))
(SETQ X (DIFFERENCE A(TIMES C D)))
```

Le espressioni SETQ *sono usate solo per grandezze numeriche?*
No, possono essere usate anche con i simboli (fig. 1.18).

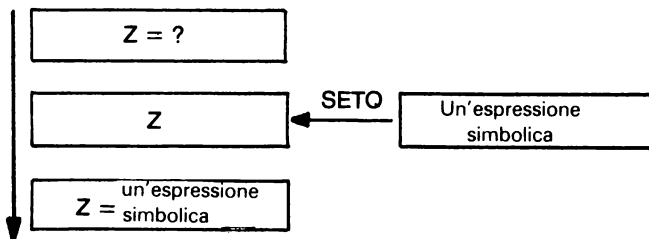



Fig. 1.18.

SIMBOLI

Oltre che un "linguaggio funzionale", che cos'è il Lisp?
Il Lisp è anche un linguaggio "simbolico". Questo significa che una delle sue caratteristiche più importanti è la capacità di manipolare dati non necessariamente numerici.

Un esempio semplice di questa capacità?
Porre il valore di una variabile, per esempio A, uguale a un altro atomo, per esempio B. Va notato che è un'operazione diversa dal dare ad A il valore B.

Che cosa fa (SETQ A B)*?*
La funzione:

(SETQ A B)

assegna ad A il valore di B. Pertanto, se B fosse stato posto uguale a 10, (SETQ A B) darebbe ad A il valore 10.

Questa espressione opera in qualche altro modo?
Sì, se B fosse stato posto uguale a J (il simbolo, non ciò per cui J sta), allora (SETQ A B) avrebbe assegnato ad A il valore J (fig. 1.19).

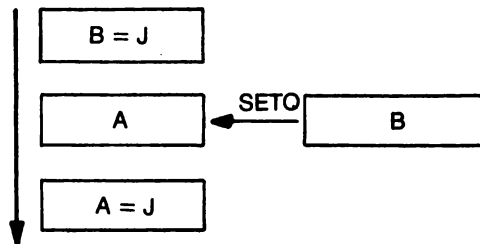



Fig. 1.19.

Esiste qualche altro modo per porre dei simboli uguali ad altri simboli?
Sì, è possibile usare la funzione QUOTE.

Se si scrivesse (SETQ A(QUOTE B)) *che cosa succederebbe?*
A sarebbe posto uguale al simbolo B.

Perché?
QUOTE (che in inglese significa letteralmente "citare") è una funzione il cui valore è il suo argomento; pertanto, il valore di (QUOTE C) è C.

In che cosa differiscono fondamentalmente (SETQ A B) *e* (SETQ A (QUOTE B))*?*
La prima S-espressione:

(SETQ A B)

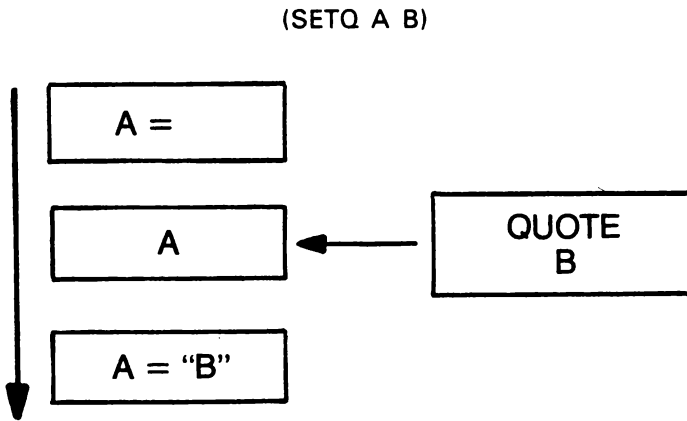



Fig. 1.20.

pone A uguale al valore di B, mentre la seconda espressione (fig. 1.20) pone A uguale al valore di (QUOTE B), che è B.

Esiste un'altra funzione simile a SETQ, *in Lisp?*
Sì, è la funzione SET.

Che cosa fa la funzione SET*?*
Nella S-espressione (SET A B), il "valore" di A è assegnato al "valore" di B. Lo si può scrivere (fig. 1.21) come:

(SET (QUOTE A) B).

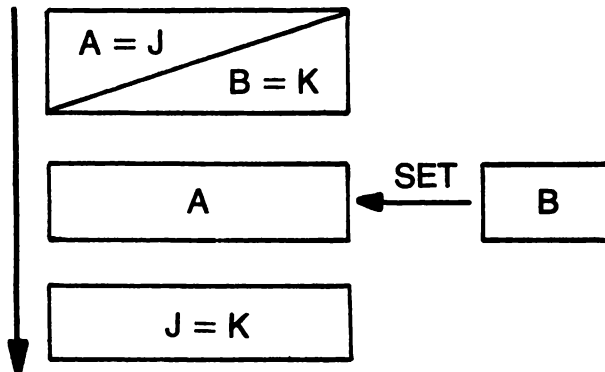



Fig. 1.21.

Che cosa succede se si usa la funzione SET *con variabili che hanno i comuni valori numerici?*
Si ottengono risultati privi di significato. Immaginiamo che A sia uguale a 10 e B sia uguale a 20 e poi scriviamo:

(SET A B).

Staremmo cercando di porre 10 uguale a 20, il che è ovviamente ridicolo.

E allora, quando si usa SET*?*
Se il valore di A fosse stato il simbolo K, allora la funzione (fig. 1.22)

(SET A B)

avrebbe posto K uguale a B.

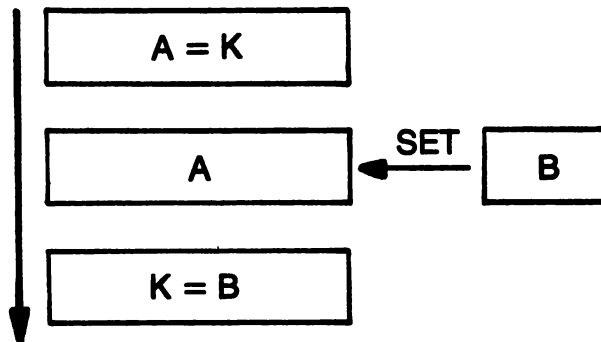



Fig. 1.22.

In Lisp i numeri hanno qualche caratteristica importante?
In Lisp ogni atomo ha un valore. Ogni numero che compare in un programma in Lisp è un atomo e il valore di questi atomi sono i numeri stessi.

È vero anche per atomi "simbolici"?
No, in genere in Lisp queste caratteristiche dei numeri non valgono anche per i simboli. Se si scrive:

(SETQ A 10)

si assegna ad A il valore di 10 — che ovviamente è 10! Ma se invece si scrive:

(SETQ A B)

si assegna ad A il valore di B. E in genere questo non è (il simbolo) B stesso.

Esistono variabili "simboliche" che stanno sempre per se stesse?
Sì, T, F e lo speciale atomo NIL. Il valore di T, che significa "vero", è T stesso. Il valore di F, che significa "falso", è NIL e il valore di NIL è NIL.

Quando si ottengono risultati senza senso con le funzioni numeriche?
Se il valore di un atomo è un altro atomo con un nome "simbolico", le funzioni numeriche daranno risultati senza valore. Ovviamente, non ha senso scrivere:

(DIFFERENCE A B)

se il valore di B è il simbolo K. Parleremo più avanti di funzioni che operano su dati simbolici.

LISTE

C'è un altro tipo di dati nel linguaggio Lisp?
La "lista" è un altro tipo di dati.

Il Lisp è un linguaggio simbolico-funzionale; è anche qualche altra cosa?
È anche un linguaggio di elaborazione di liste. In effetti il nome Lisp deriva dall'espressione inglese "List Processing".

Che cos'è una lista in Lisp?
Una lista è un insieme ordinato di dati, simile in un certo senso a una matrice (fig. 1.23).
Questa S-espressione:

(FORMAGGIO 2 3 4 CASA 67 RAGAZZA)

è una S-espressione di una lista di 7 "elementi".

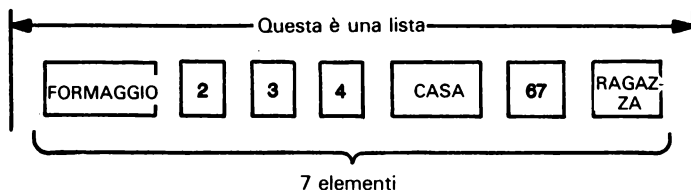



Fig. 1.23.

In che cosa differiscono una lista in Lisp e una matrice in altri linguaggi?
Una lista in Lisp non può essere "indicizzata" come una matrice. Se ci si vuole riferire o indirizzare al terzo elemento nella S-espressione data (il numero 3, in questo caso), non si può semplicemente mettere il numero 3 in un registro indice o nel suo analogo e pensare di poter avere immediatamente il terzo atomo. Questo dipende dal modo in cui gli atomi in una lista (o i relativi riferimenti) sono "concatenati" insieme.

Come sono concatenati gli atomi?
Tutti gli atomi, o riferimenti, hanno un puntatore associato, che "punta" al riferimento successivo (fig. 1.24). Questo, chiaramente, significa che gli indirizzi dei vari atomi e riferimenti non sono in "sequenza" come sarebbero in una matrice, per esempio in Basic o in Fortran. Ma vedremo la cosa più in dettaglio in seguito.

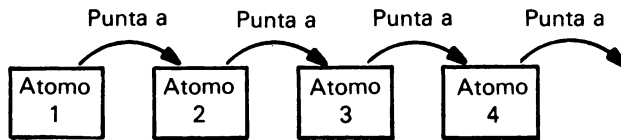



Fig. 1.24.

Che cosa succede se si scrive (SETQ A (FORMAGGIO 234 CASA 67 RAGAZZA))*?*
Vorrebbe dire cercare di porre A uguale alla lista indicata, e si incapperebbe in qualche problema (fig. 1.25).

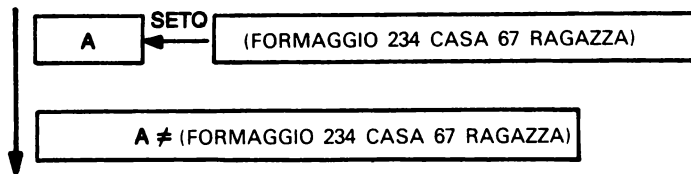



Fig. 1.25.

Che genere di problemi sorgono con questa forma di manipolazione?
Prendiamo l'esempio:

(SETQ A (DIFFERENCE B C))

È ovvio che si vuole porre A uguale a B – C. Quello che non si vuole è porre A uguale alla "lista" (DIFFERENCE B C). Pertanto è stata formulata una convenzione che in Lisp vale sempre (fig. 1.26).

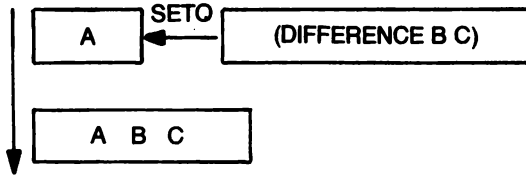



Fig. 1.26.

Qual è questa "convenzione"?
Ogniqualvolta una espressione (come (DIFFERENCE B C)) appare all'interno di una S-espressione, il "primo" atomo incontrato (in questo esempio DIFFERENCE) viene interpretato come la specificazione di una funzione. Gli atomi successivi sono intesi come argomenti di tale funzione. Ovviamente, questo vale anche per parentesi "stratificate" (parentesi entro altre parentesi). Se si presenta una sottoespressione, il primo atomo viene "ancora" inteso come funzione, e via di seguito (fig. 1.27).

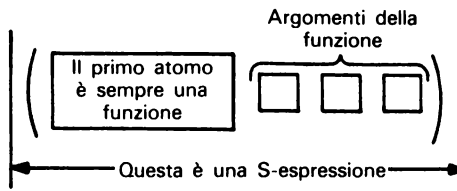



Fig. 1.27.

Il problema ancora non è chiaro. In che cosa consiste?
Nel primo esempio:

(SETQ A(FORMAGGIO 234 CASA 67 RAGAZZA))

il primo atomo nella sottoespressione è FORMAGGIO e il Lisp vedrebbe FORMAGGIO come una funzione. Poiché non esiste una funzione chiamata FORMAGGIO, il sistema operativo del Lisp dichiarerebbe un errore!

E allora come bisogna scrivere (SETQ A(FORMAGGIO 234 CASA 67 RAGAZZA)) *in modo che sia una espressione lecita?*
Si usa la funzione QUOTE (fig. 1.28). Se si scrive:

(SETQ A(QUOTE(FORMAGGIO 234 CASA 67 RAGAZZA)))

non si hanno problemi. Nella funzione QUOTE (e solo nella funzione QUOTE) la regola del "primo" atomo non viene seguita.
L'atomo FORMAGGIO non viene preso come un nome di funzione. Il

valore della funzione QUOTE in questo esempio, come in tutti gli usi di questa funzione, è l'argomento della funzione QUOTE *esattamente come si presenta* (viene cioé inteso come una stringa letterale).

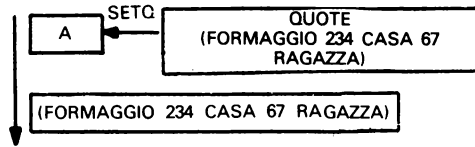



Fig. 1.28.

Esiste un altro modo per produrre una lista come valore di una funzione?
Sì, con l'uso della funzione LIST (fig. 1.29). Il valore di:

(LIST FORMAGGIO RAGAZZA)

è (FORMAGGIO RAGAZZA). Questo risultato è ovviamente identico al valore di:

((QUOTE(FORMAGGIO RAGAZZA))

ma esiste una differenza fra le funzioni LIST e QUOTE.

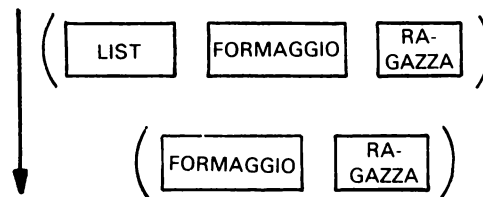



Fig. 1.29.

In che cosa differiscono LIST e QUOTE?
Se gli argomenti sono simbolici, esiste una forte differenza fra LIST e QUOTE (fig. 1.30). Assumiamo che A abbia il valore 3. Allora:

(LIST D A C A A)

ha il valore di:

(D 3 C 3 3)

ma se si fosse usata invece la funzione QUOTE:

(QUOTE (D A C A A))

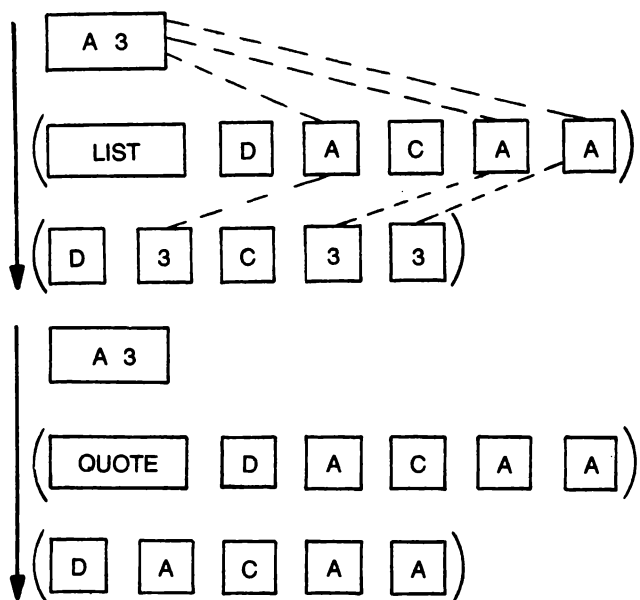



Fig. 1.30.

si sarebbe ottenuto il valore:

(D A C A A).

Qualche funzione può avere come proprio valore una lista?
Sì, in particolare, c'è una funzione aritmetica che ha come valore una lista di due atomi. È la funzione DIVIDE e la sua lista è data dal quoziente e dal resto.

(DIVIDE 35 4)

ha come proprio valore la lista (8 3). Tuttavia, una lista del genere non può essere usata direttamente dalle funzioni aritmetiche, anche qualora contenesse un solo "elemento". Pertanto, si può vedere facilmente che (DIFFERENCE S 6) è assolutamente senza senso (e provoca una segnalazione di errore dal sistema Lisp) se S è la lista (7), ma è uguale a 1 se S è l'atomo 7.

SOTTOLISTE

Qual è il risultato del concatenamento degli elementi in memoria?
A seconda del modo in cui avviene il concatenamento, gli elementi di una lista possono essere atomi o sottoliste.

Che cos'è una sottolista in Lisp?
Una sottolista è semplicemente una lista contenuta in un'altra lista (fig. 1.31). Per esempio, si può voler costruire una lista di quattro elementi, 1

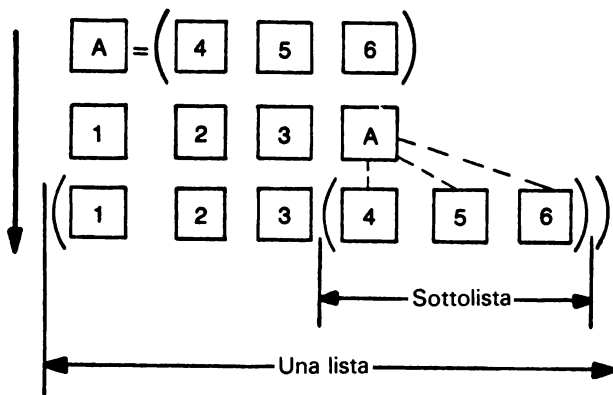



Fig. 1.31.

2 3 e A, dove A sta per un'altra lista di 3 elementi, 4 5 e 6. Per questo caso scriviamo:

(1 2 3(4 5 6))

come S-espressione di questa lista. Ovviamente anche le sottoliste di una lista possono contenere ulteriori sottoliste, e via all'infinito.

Qual è la relazione fra il Lisp come linguaggio di "elaborazione di liste" e il Lisp come linguaggio funzionale?
Le espressioni funzionali, o S-espressioni, che specificano le chiamate di funzione in Lisp specificano anche liste. Pertanto, quella che segue

(SETQ A(PLUS B (TIMES C 10)))

è una lista di tre elementi. Il primo elemento è SETQ, il secondo è A e il terzo è a sua volta una lista di tre elementi. Questi tre elementi sono PLUS, B e un'altra lista di tre elementi formata da TIMES, C e 10.

Come appare in memoria un riferimento di funzione in Lisp?
Appare in memoria come una lista. Ogniqualvolta in un programma in Lisp immesso in memoria compare una S-espressione come (DIFFERENCE X Y), si forma una lista. In questo caso è la lista dei tre elementi DIFFERENCE, X e Y.

Normalmente, che cos'è il primo elemento di queste liste?

Il primo elemento di una lista del genere normalmente sarà un atomo, dal momento che sta per una funzione. Tuttavia vi sono casi in cui in questa posizione, per specificare una funzione, si può usare una sottolista.

Come sono chiamati gli "elementi" restanti?
Gli elementi che seguono il primo sono o atomi o sottoliste. Una sottolista in questa posizione specifica una funzione all'interno della funzione originale.

Come viene rappresentata in memoria una funzione?
Un riferimento di funzione viene rappresentato in memoria come una "lista".

Qual è la regola generale per le liste e le S-espressioni?
Come regola generale, "a ogni S-espressione in Lisp corrisponde una lista e a ogni lista in Lisp corrisponde una S-espressione" (fig. 1.32).

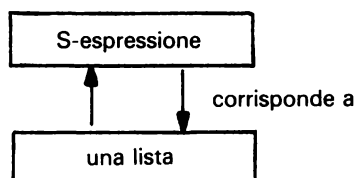



Fig. 1.32.

RICORSIONE

Il Lisp dunque è un "linguaggio di elaborazione di liste funzionale e simbolico". È anche qualche cosa d'altro?
Il Lisp è anche un linguaggio ricorsivo.

È ricorsivo come Algol?
In Algol esistono funzioni che in effetti richiamano se stesse, direttamente o indirettamente (fig. 1.33). Se la funzione X contiene entro la sua struttura una chiamata a se stessa, allora quella X chiama se stessa direttamente. Se una funzione X chiama un'altra funzione di nome C e questa seconda funzione C chiama X, allora X chiama se stessa indirettamente.

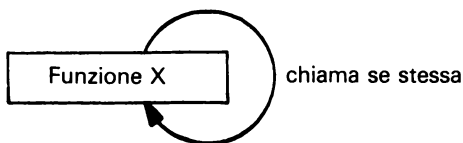



Fig. 1.33.

Si può dare un enunciato più generale?
Sì. Se vi sono in Algol varie funzioni chiamate Z_1, Z_2, Z_3, ..., tali che Z_i chiama Z_{i+1} per $1 \leq i \leq n$ e Z_n chiama Z_1, allora Z_1 chiama se stessa indirettamente (e lo stesso vale, in effetti, per tutte le altre Z_i) (fig. 1.34).

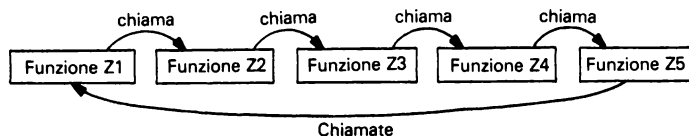



Fig. 1.34.

Vale lo stesso per il Lisp?
In Lisp una funzione definita può essere ricorsiva come le funzioni dell'Algol, ma in Lisp si usa anche il concetto di funzione ricorsiva come è definito nella logica.

Che cos'è una definizione "circolare"?
Prendiamo l'esempio del fattoriale. Quando si usa la definizione standard di fattoriale, che dice che il fattoriale di *n* è uguale a *n* moltiplicato per il fattoriale di *n*–1, si dà una definizione circolare. Si definisce una funzione usando una definizione che include la funzione stessa. Tipicamente, questo renderebbe l'idea di una definizione impropria. Nell'esempio del fattoriale, se si specifica o si definisce che il fattoriale di 0 è 1, la definizione dà il valore del fattoriale per qualunque intero positivo.

Come si chiamano le funzioni che sembrano "circolari"?
Le definizioni che sembrano circolari, ma di fatto specificano una funzione in modo univoco su un certo intervallo, sono dette definizioni di una funzione ricorsiva.

Una funzione può avere più di una definizione?
Sì, molte funzioni hanno sia definizioni comuni, sia definizioni ricorsive.

Questa idea di ricorsività può aiutarci a definire la S-espressione?
Sì, si può usare una definizione ricorsiva di S-espressione. Una S-espressione è o un atomo o una parentesi aperta seguita da una se-

quenza di S-espressioni separate da spazi vuoti o da virgole e seguita da una parentesi chiusa. Le S-espressioni della sequenza a loro volta possono essere atomi o altre sequenze.
Per esempio:

((A B)C((D E) F))

Questa S-espressione è costituita da una parentesi aperta seguita dalla sequenza di S-espressioni (A B), C, ((D E)F) seguita da una parentesi chiusa. Ciascuna di queste sequenze può essere identificata come una S-espressione nello stesso modo.

Usando le definizioni ricorsive, che cos'è una lista?
Una lista è una sequenza di riferimenti, ciascuno dei quali può essere un riferimento a un atomo o a un'altra lista (fig. 1.35). In riferimento a questa definizione ricorsiva di lista, è importante ricordare che una lista non è una sequenza di atomi. La linea di demarcazione nelle definizioni qui diventa molto sottile, ma per il Lisp è molto importante. Un atomo è rappresentato in memoria da un elemento che comprende il suo nome e che può includere anche varie proprietà dell'atomo. Una lista che include questo atomo "include" solamente il puntatore a quell'atomo.

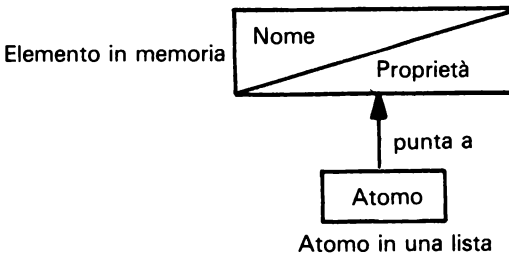



Fig. 1.35.

Qual è allora la definizione di lista?
In Lisp una lista è una lista di coppie. Un elemento di una coppia è un puntatore a un atomo mentre l'altro è un puntatore alla coppia successiva (fig. 1.36).

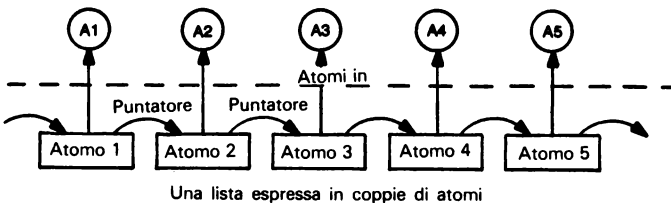



Fig. 1.36.

IL SISTEMA LISP

Che cos'è il sistema operativo Lisp?
La maggior parte dei sistemi Lisp consente di effettuare qualche esperimento senza doversi preoccupare di funzioni di programmi. Ovviamente, si possono usare le funzioni "standard" già incorporate come SIN, COS, DIFFERENCE, PLUS e altre.

Il funzionamento del sistema Lisp è analogo a quello del Fortran?
No, normalmente quello del Fortran è considerato un compilatore. Il programma sorgente viene immesso nel sistema e il compilatore Fortran compila o "crea" un programma "oggetto". Quando si fa girare un programma in Fortran quello che viene eseguito è in effetti il programma oggetto. In Lisp viene usato un interprete. Si immette nel sistema Lisp una funzione, e il suo valore viene calcolato immediatamente. Questo non significa che non esistano compilatori anche per il Lisp, ma di questo parleremo più avanti.

Considerando come funziona, che cosa si può fare con il sistema interprete?
Non abbiamo ancora imparato a programmare in Lisp, quindi discuterò solo le idee di base e come realizzare funzioni semplici. Se si batte alla tastiera di un qualunque terminale:

```
TIMES (5 6)
```

il computer risponderà con:

```
30.
```

La riga che hai appena battuto sembra sbagliata. Perché invece dà il risultato giusto?
Il sistema Lisp non legge solo una S-espressione alla volta, ma due S-espressioni alla volta (fig. 1.37). La prima S-espressione specifica una funzione, mentre la seconda specifica i suoi argomenti. Pertanto, quando il sistema Lisp legge TIMES (5 6), legge in effetti due S-espressioni. La prima è la funzione TIMES e la seconda dà i suoi argomenti, 5 e 6. È immediato vedere che TIMES (5 6) può essere derivata dall'espressione propria (TIMES 5 6) come segue. Si prenda la prima parentesi aperta e la si sposti in avanti, in modo che appaia *dopo* il primo elemento della S-espressione, anziché davanti. Questo crea un "cortocircuito".

Questo cortocircuito vale anche per altre, magari per tutte le funzioni?
No, se si prende l'esempio di:

Fig. 1.37.

TIMES (5 PLUS (6 7))

come riga in ingresso al sistema Lisp, l'interprete segnalerà un errore di input.

Perché?
La lista (5 PLUS (6 7)) è una lista di tre elementi, l'intero 5, il simbolo PLUS e la lista (6 7). Pertanto, la forma PLUS (6 7) non può essere usata all'interno di un'altra coppia di parentesi. Ricontrollate la domanda "che cos'è questa convenzione?" nel paragrafo sulle liste.

Quando si può usare questo cortocircuito?
Lo si usa con la modifica seguente. Ogniqualvolta si usa una funzione al livello "più alto" (non inclusa fra parentesi), si può usare un formato come TIMES (5 6) o in generale "FUNZIONE" (A B). Ma quando si usa una funzione a un livello diverso (in altre parole quando è racchiusa fra parentesi) si deve usare il formato convenzionale: (FUNZIONE A B).

C'è ancora qualcosa da dire a proposito di questo "livello più alto"?
Sì. Il livello più alto è solo un caso particolare del concetto di livelli di parentesi (fig. 1.38). Un atomo entro una S-espressione a qualche livello di parentesi si può dire "al livello n". Se si trova all'esterno delle parentesi è al "livello 0".

Fig. 1.38

Tutte le funzioni del Lisp viste fin qui possono operare al livello 0?
Sì, tutte possono operare al livello 0, fatta eccezione per la funzione SETQ. Per questa funzione è necessario un livello 1 o superiore.

Si può avere qualche esempio, considerando solo il livello 0?
Si può scrivere:

```
TIMES (5 6)
30
DIFFERENCE (45 30)
15
PLUS (23 34)
57
QUOTIENT (10 2)
5
REMAINDER (11 2)
1
```

È evidente che i numeri che seguono gli esempi sono le soluzioni, o i risultati delle S-espressioni, forniti in risposta dal sistema Lisp.

Vi sono funzioni senza argomenti?
Sì, esistono funzioni senza argomenti, anche se non le abbiamo ancora citate.

Come si potrebbe scrivere una funzione del genere, usando il cortocircuito in Lisp?
Se avessimo una funzione del genere chiamata AABBCC, potremmo scrivere:

AABBCC().

Per che cosa sta il simbolo ()?
I simboli () stanno per una lista vuota (una lista che non contiene elementi).

C'è qualche altro modo per denotare una lista vuota?
Sì, si può produrre una lista vuota scrivendo NIL. NIL ha vari significati in Lisp, come si è già detto.

Come si usa il simbolo NIL*?*
Con la nostra immaginaria funzione AABBCC, scriveremmo:

AABBCC NIL

che ha lo stesso significato di:

AABBCC().

SOTTOESPRESSIONI

Si possono scrivere funzioni più complesse, usando sempre l'interprete, senza programmare?
Sì, ipotizziamo di voler scrivere l'espressione:

(TIMES(DIFFERENCE 20 10) (PLUS 5 5))

Spereremmo che il sistema ci dia una risposta 100, ma, come abbiamo già detto, non si può semplicemente scrivere o immettere questa espressione funzionale e aspettarsi che funzioni.

Perché?
Perché TIMES è una funzione al livello più alto. Nella maggior parte degli interpreti Lisp, c'è un altro motivo per cui non si può semplicemente scrivere:

(TIMES(DIFFERENCE 20 10)(PLUS 5 5))

anche se, dal paragrafo precedente sembrerebbe possibile. Questo perché la *convenzione* (citata in precedenza) relativa alla valutazione di funzioni all'interno di altre funzioni non viene seguita quando si tratta con funzioni a livello di parentesi 0. Quel che abbiamo scritto equivale a moltiplicare la lista (DIFFERENCE 20 10) per la lista (PLUS 5 5) e non i numeri 20-10 e 5+5. La funzione TIMES opera su numeri e non su liste. Di conseguenza, l'interprete Lisp genererà un errore di sistema.

Come si possono valutare queste espressioni?
Il problema può essere aggirato usando la funzione EVAL. Se A è una qualunque S-espressione che coinvolge interi, come nell'esempio precedente, si costruisce la riga da immettere nel sistema nella forma EVAL(A). Alcuni sistemi Lisp usano diversi tipi di EVAL. Sono:

EVAL(A)
EVAL(A NIL)

oppure

EVAL (A)

Poiché A è racchiuso fra parentesi, non è necessario (e sarebbe *errato*) spostare la prima parentesi alla destra del nome di funzione, come avevamo detto nella sezione sul sistema Lisp. Pertanto, per valutare l'espressione:

(TIMES(DIFFERENCE 20 10)(PLUS 5 5))

si deve battere:

EVAL((TIMES(DIFFERENCE 20 10)(PLUS 5 5)))

e il sistema risponderà 100.

Esiste un altro metodo per valutare le funzioni, simile all'uso della funzione EVAL?
Sì, si può usare il formato (LAMBDA NIL A) NIL. Questo metodo funziona su tutti i sistemi Lisp indipendentemente dal tipo.

Come si usa la funzione LAMBDA?
Per valutare l'espressione descritta prima mediante l'uso della funzione LAMBDA, si deve battere:

(LAMBDA NIL(TIMES(DIFFERENCE 20 10)(PLUS 5 5)))NIL

che ovviamente ci darà in risposta il valore 100.

Quanto è estesa la funzione LAMBDA, *rispetto a qualunque altra funzione?*
Qualunque S-espressione aritmetica può essere valutata mediante la funzione LAMBDA, indipendentemente dalla complessità dell'S-espressione stessa.

Si possono usare parentesi extra nel Lisp?
No! Dovrebbe essere facile capire che non si possono inserire parentesi extra senza un motivo specifico (fig. 1.39).

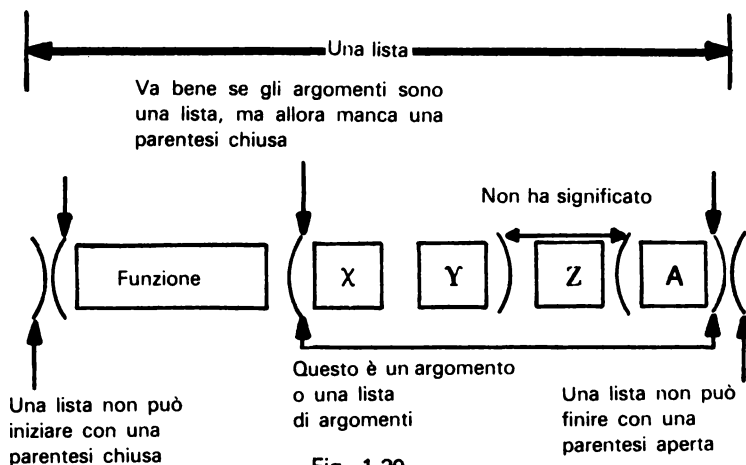



Fig. 1.39.

È corretto scrivere EVAL((PLUS(8)(DIFFERENCE 34 25)))? *Può girare?*
No, sarebbe sicuramente scorretto, perché (8) significherebbe che 8 è una funzione senza argomenti (fig. 1.40).

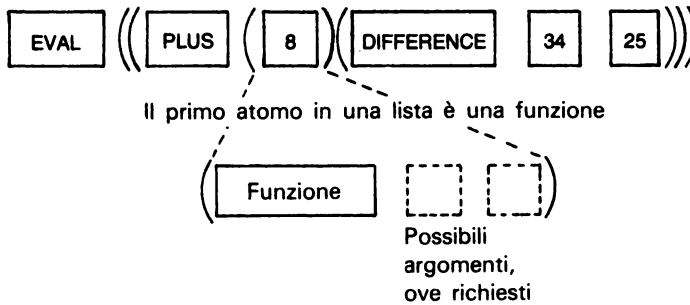



Fig. 1.40.

RIPASSO

FORMAGGIO *è un atomo?*
Sì, perché FORMAGGIO è una stringa di caratteri.

9876 *è un atomo?*
Sì, perché 9876 è una stringa di caratteri.

K *è un atomo?*
Sì, perché K è una stringa di caratteri, lunga un solo carattere.

(IL FORMAGGIO E' BUONO) *è una lista?*
Sì, perché è un insieme di atomi racchiusi fra parentesi.

(RAGAZZA) *è una lista?*
Sì, perché RAGAZZA è un atomo ed è racchiuso fra parentesi.

((I TOPI SONO)SIMPATICI) *è una lista?*
Sì, perché le due S-espressioni sono racchiuse fra parentesi.

ABC *è una S-espressione?*
Sì, perché tutti gli atomi sono S-espressioni.

(A B C) *è una S-espressione?*
Sì, perché è una lista.

(A(B)C) *è una S-espressione?*
Sì, perché tutte le liste sono S-espressioni.

Quante S-espressioni vi sono nella lista (GIOVANNI MANGIA INSALATA)*?*
Tre: GIOVANNI, MANGIA, INSALATA.

() *è una lista?*
Sì, perché è un insieme vuoto di S-espressioni racchiuso fra parentesi. È una speciale S-espressione chiamata lista vuota.

(0 0 0) *è una lista?*
Sì, perché è un insieme di S-espressioni racchiuso fra parentesi.

Come si scrive, in Lisp, 4COSA – 6*?*
(DIFFERENCE (TIMES 4(COS A)) 6).

Come si scrive $A^2/B^3 - V^2$ *in Lisp?*
(QUOTIENT(TIMES A A)(DIFFERENCE(TIMES B B B) (TIMES V V))).

Che cosa significa, in Lisp, (TIMES A S D (MINUS F))*?*
A*S*D*-F o, se si preferisce, A*S*D*(-F).

Che cosa significa, in Lisp, (REMAINDER A B)*?*
Questa funzione dà il resto che si ottiene dividendo A per B.

Quali delle seguenti sono S-espressioni?

```
(TIMES(MINUS B)(MINUS C)
(PLUS A C H T R))
(QUOTIENT)PLUS A B)(DIFFERENCE G H))
(REMAINDER (PLUS A B C D)(TIMES F G H))
```

Le prime tre non sono S-espressioni valide, ma l'ultima sì. Nella prima espressione manca una parentesi chiusa alla fine. La seconda espressione ha una parentesi chiusa di troppo alla fine. La terza espressione ha la seconda parentesi "aperta" rovesciata.

Si scriva nella notazione del Lisp il seguente enunciato di assegnazione: R *è uguale a* $(P-T)/(Y^2)$.
(SETQ R (QUOTIENT(DIFFERENCE P T)(TIMES Y Y)))).

A questo punto, c'è ancora qualcosa di cui non abbiamo parlato?
Se avete ancora dei dubbi su ciò che abbiamo detto fin qui, rileggetevi i paragrafi precedenti.

CAPITOLO DUE

Funzioni e un po' di logica

Che cosa significa in Lisp la parola DEFINE*?*
La parola o simbolo DEFINE ("definisci" in inglese) significa che quel che segue è la definizione di una funzione.

DEFINE *è una "funzione"?*
Sì, come abbiamo detto prima, il Lisp è fatto solo di funzioni; non ci sono costruzioni di altro tipo.

DEFINE *ha un analogo in Fortran?*
Sì, la funzione DEFINE assomiglia sotto alcuni aspetti alla parola FUNCTION del Fortran. Ma in Lisp DEFINE è una funzione mentre FUNCTION in Fortran è una *parola riservata.*

Quali sono gli argomenti della funzione DEFINE*?*
DEFINE ha un solo argomento ed è una lista di funzioni che debbono essere definite. Ciascuna funzione che deve essere definita è a sua volta una lista di due *elementi*, il primo dei quali è il nome della funzione e il secondo è la definizione o la descrizione della funzione. Pertanto, scriviamo:

```
DEFINE(((AAAz1) (BBBz2) (CCCz3)))
```

dove z1, z2 e z3 stanno per tre descrizioni o definizioni di funzione. Questo esempio definirebbe le tre funzioni AAA, BBB e CCC rispettivamen-

te (fig. 2.1). L'espressione ((AAAz_1) (BBBz_2) (CCCz_3)) è la lista di argomenti di DEFINE. Tuttavia, DEFINE ha un solo argomento e, in effetti, ((AAAz_1) (BBBz_2) (CCCz_3) è un argomento solo. Pertanto, la definizione di DEFINE continua a valere.

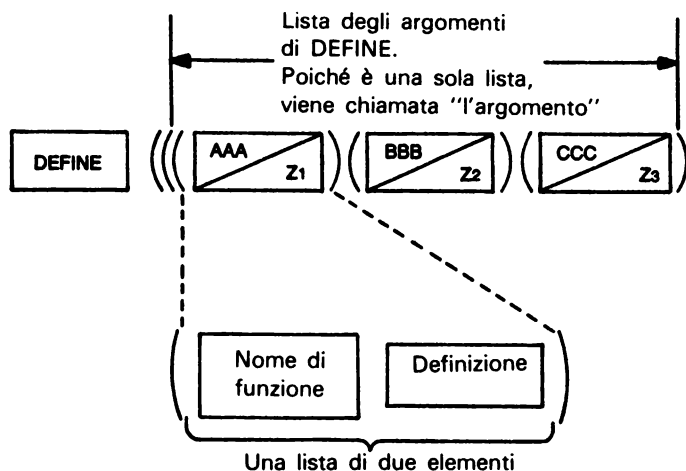



Fig. 2.1

Ma le funzioni da definire sono tre! Come è possibile che ci sia un argomento solo?
È possibile perché l'argomento stesso è scritto come una lista di tre S-espressioni. Nota che nell'esempio dato AAA, BBB e CCC stanno per, o sono assunte a significare che stanno per, i nomi AAA, BBB e CCC, e non le grandezze per cui AAA, BBB, CCC stanno.

Quali sono le specifiche legittime per una definizione di funzione in Lisp?
Qualunque specifica legittima di una funzione già nota può essere usata nella costruzione o definizione di nuove funzioni in Lisp, se si usa la funzione DEFINE.

Che cosa rappresenta l'espressione che segue?

```
DEFINE (((AAA PLUS)(BBB REMAINDER)(CCC QUOTIENT)))
```

Hai definito AAA in modo che sia la stessa funzione di PLUS, BBB in modo che sia identica a REMAINDER e CCC in modo che coincida con QUOTIENT (fig. 2.2). Pertanto se poi scrivi:

```
(AAA C B)
```

sommi C e B tra loro come se usassi l'espressione:

(PLUS C B).

E analogamente se scrivi

(BBB R T)

trovi il resto della divisione di R per T esattamente come con l'espressione:

(REMAINDER R T)

o ancora con

(CCC 4 2)

trovi il quoziente di 4 e 2, come se avessi usato la funzione standard QUOTIENT:

(QUOTIENT 4 2).

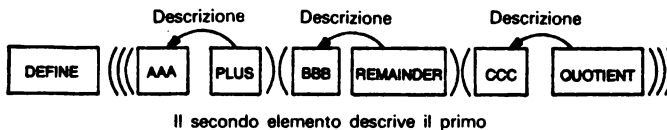



Fig. 2.2.

Si possono combinare queste funzioni?
Sì, non appena una funzione è stata definita, opera come qualunque altra funzione e può essere usata come tale.

Si può scrivere una espressione più complessa usando le funzioni AAA, BBB *e* CCC*?*
Assumendo che AAA stia per PLUS, BBB per REMAINDER e CCC per QUOTIENT, si può scrivere:

(BBB(CCC(AAA 44 2)(AAA 1 1))7)

che ci darà lo stesso risultato dell'espressione:

(REMAINDER (QUOTIENT (PLUS 44 2)(PLUS 1 1))7)

e ambedue, essendo identiche come comportamento, ci danno il valore 2 (fig. 2.3).

BBB
CCC
AAA
44
2
AAA
1
1
7
ha lo stesso significato di
REMAINDER
ha lo stesso significato di
QUOTIENT
ha lo stesso significato di
PLUS
ha lo stesso significato di
PLUS

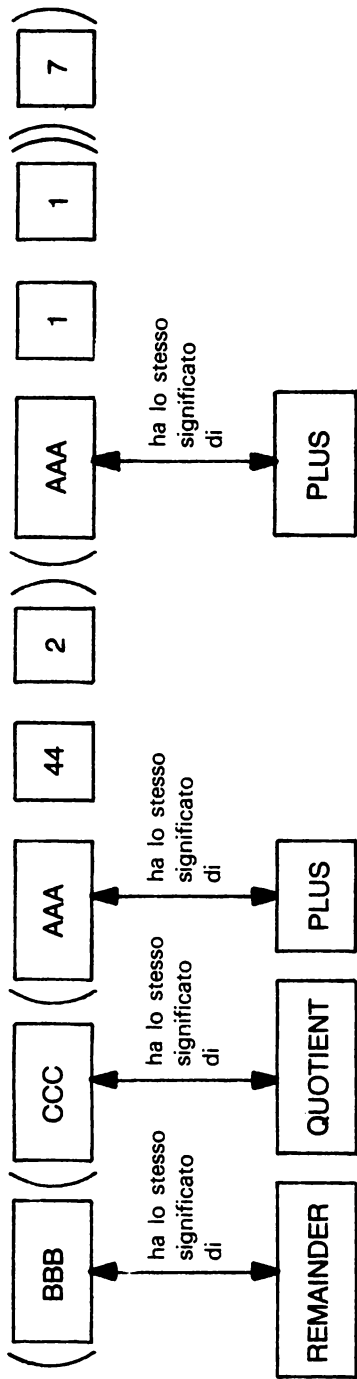



Fig. 2.3.

Che cosa è importante, quando si usa la funzione DEFINE?
È evidente che si possono definire nuove funzioni solo con le funzioni standard e con funzioni che sono state già definite in precedenza.

LE PIÙ COMUNI FUNZIONI DEL LISP

Con l'elenco delle funzioni più comuni nel Lisp, nelle definizioni e negli esempi, useremo queste abbreviazioni:

A	al posto di	atomo
C	»	"corpo" del programma
F	»	funzione
L	»	lista
NIL	»	falso
S	»	S-espressione
T	»	vero ("*true*")
X	»	numero
AR	»	argomento (qualunque dei precedenti).

Le notazioni sono numerate come segue: 1 per la prima posizione, 2 per la seconda posizione in una lista e 1 ... n sta per qualunque numero di argomenti della classe specificata dalla lettera che precede.

funzione	*definizione*	*esempio*
ABS	Dà il valore assoluto dell'argomento	(ABS X)
ADD1	Dà l'argomento maggiorato di 1	(ADD1 X)
AND	Dà NIL se qualcuna delle S-espressioni è NIL, dà T in caso contrario	(AND ... S...S) (fig. 2.4)
APPEND	Dà come risultato un'unica lista i cui elementi sono gli elementi delle due liste che costituiscono i suoi argomenti	(APPEND L1 L2)
APPLY	Agisce sugli argomenti con la funzione data come se tale funzione apparisse per prima nella S-espressione	(APPLY F AR)
ASSOC	Usa il suo primo argomento come una chiave e cerca la stessa chiave nella lista che è	(ASSOC A L)

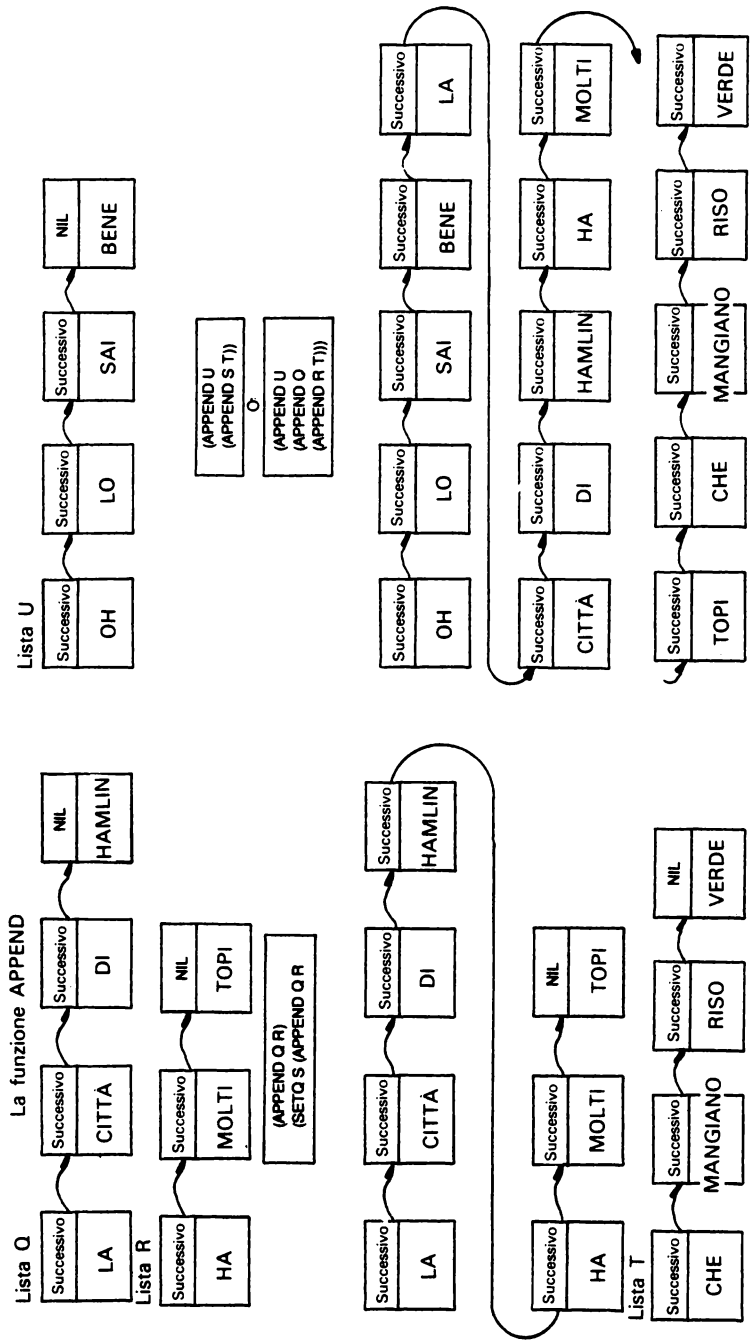



Fig. 2.4. La funzione APPEND.

	il suo secondo argomento. Il valore di ASSOC è tutto l'elemento il cui CAR corrisponde alla chiave. Se la chiave non viene trovata, il valore è NIL	
ATOM	Dà il valore T se la S-espressione è un atomo, altrimenti dà NIL	(ATOM S)
CAR	Dà il primo elemento di una lista	(CAR L) (fig. 2.5)
CDR	Dà la lista meno il primo elemento	(CDR L) (fig. 2.8)
COND	La funzione di diramazione del Lisp	(COND L1 L2 Ln)
CONS	Aggiunge la S-espressione data alla lista come suo primo elemento	(CONS S L) (fig. 2.9)
DEFINE	Definisce una funzione, e ha come valore il nome della funzione	(DEFINE F AR1 ...ARn ... B)
DEFPROP	Memorizza la S-espressione come proprietà data dell'atomo	(DEFPROP A S P)
DELETE	Elimina le occorrenze della S-espressione che appaiono come elementi di unà lista. Il numero delle occorrenze che vengono eliminate è dato dal terzo argomento. Se questo argomento manca, vengono eliminate tutte le occorrenze	(DELETE S L X) (fig. 2.10)
DIFFERENCE	Dà il primo argomento meno il secondo	(DIFFERENCE X X)
DO	La funzione di iterazione in Lisp	DO
EQUAL	Dà il valore T se le S-espressioni sono "identiche"	(EQUAL S S)
EVAL	Dà il valore della S-espressione valutata	(EVAL S)
EXPLODE	Dà una lista di atomi costituiti ciascuno da un carattere a partire dall'atomo dato	(EXPLODE A) (fig. 2.11)
EXP	Eleva *e* alla potenza indicata	(EXP X)

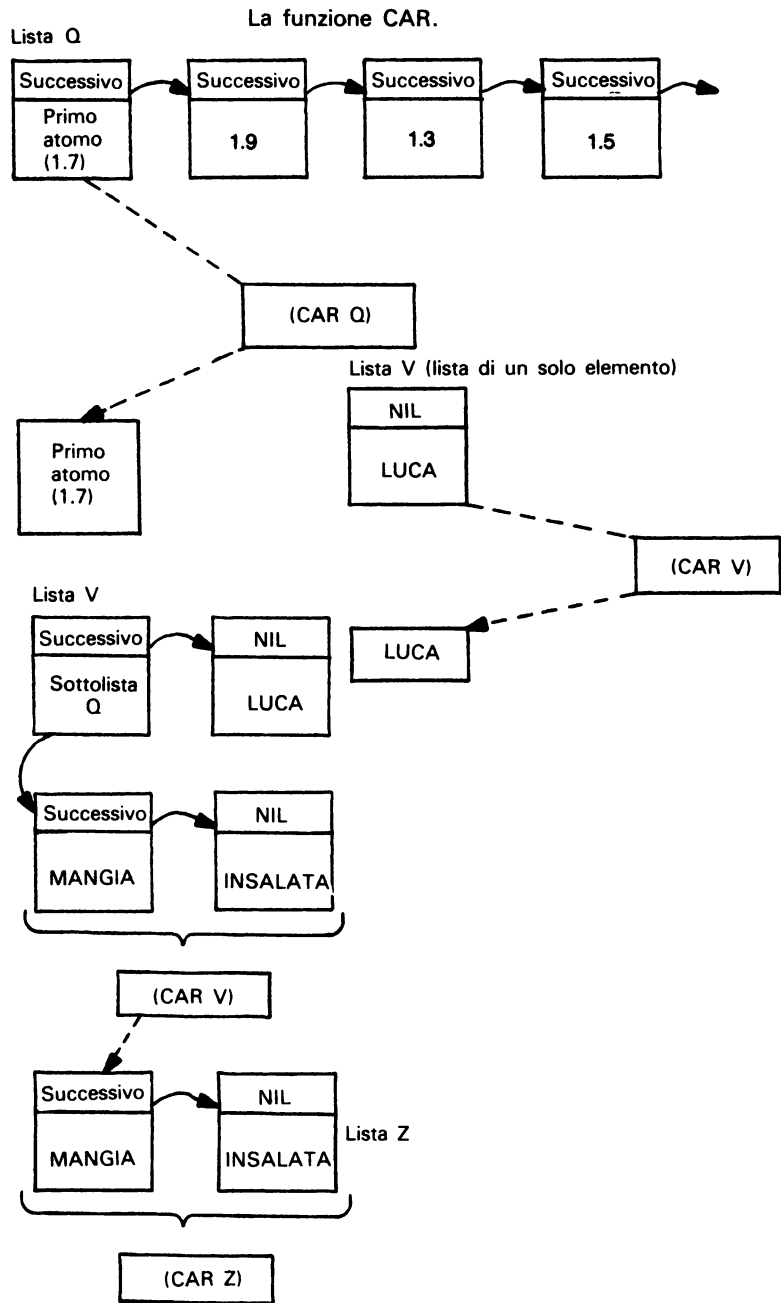



Fig. 2.5. La funzione CAR.

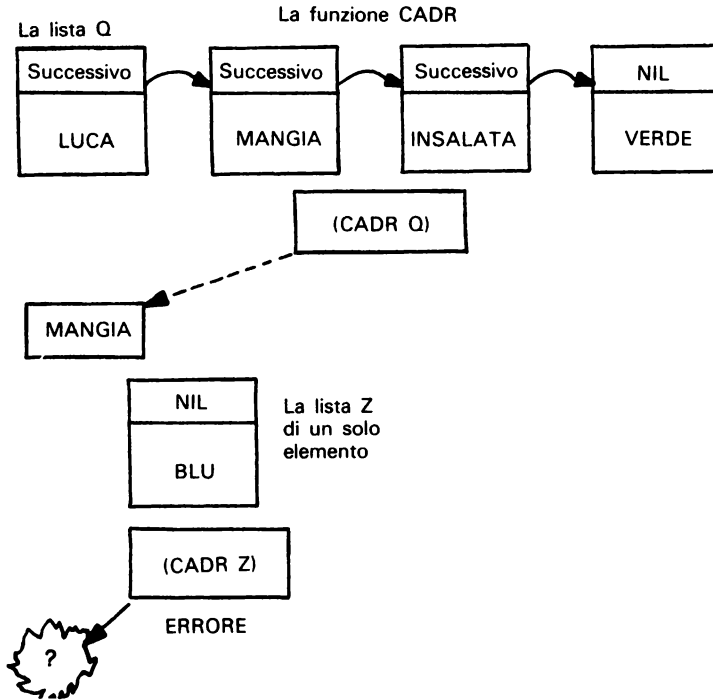



Fig. 2.6. La funzione CADR.

EXPT	Eleva il primo argomento alla potenza indicata dal secondo argomento	(EXPT X X)
FUNCALL	Opera sugli argomenti con la funzione data (un po' come APPLY)	(FUNCALL F AR1 ... ARn)
GET	Dà il valore della proprietà citata, associata all'atomo	(GET A P)
GO	Salto incondizionato	(GO T)
GREATERP	Dà T se tutti gli argomenti sono in ordine decrescente	(GREATERP X1 ... Xn)
IMPLODE	Riunisce i singoli caratteri dati in una lista in forma di atomo	(IMPLODE L) (fig. 2.12)
LAMBDA	Simile a DEFINE	(LAMBDA AR B)
LAST	Dà la lista con tutti gli elementi prima della rimozione dell'ultimo	(LAST L)
LENGTH	Dà il numero degli elementi in una lista	(LENGTH L) (fig. 2.13)

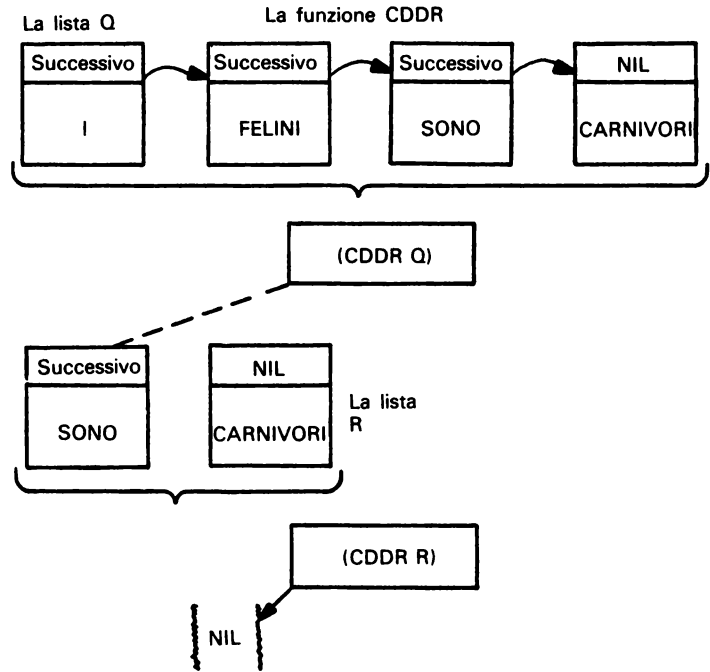



Fig. 2.7. La funzione CDDR.

LESSP	Dà T se tutti gli argomenti sono in ordine crescente	(LESSP X1...Xn) (fig. 2.14)
LIST	Dà una lista di *n* elementi costruita a partire dagli *n* elementi dati	(LIST S1...Sn) (fig. 2.15)
MAPCAR	Applica una funzione a una lista o a una lista di argomenti. Dà una lista dei risultati	(MAPCAR F AR1 ...ARn) (fig. 2.16)
MAX	Dà il più grande fra gli argomenti	(MAX X1...Xn) (fig. 2.17)
MEMBER	Dà T se la S-espressione è uguale a un elemento di massimo livello (livello 0) della lista	(MEMBER S L)
MIN	Dà il più piccolo fra gli argomenti	(MIN X1...Xn) (fig. 2.18)
MINUS	Dà il "meno unario" (il negativo) dell'argomento	(MINUS X)
MINUSP	Dà T se l'argomento è negativo	(MINUSP X)

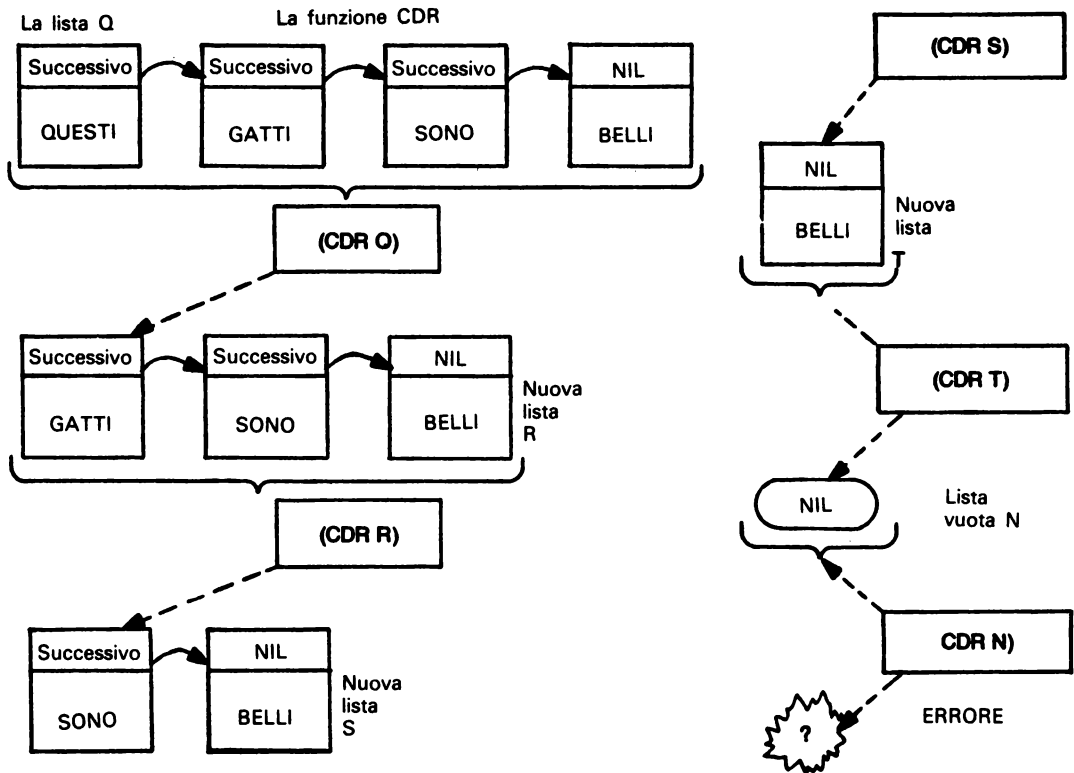



Fig. 2.8. La funzione CDR.

NOT	Dà T se la S-espressione è NIL e NIL se è qualche altra cosa (come NULL)	(NOT S)
NULL	Dà T se la S-espressione è una lista vuota e NIL in caso contrario (come per NOT)	(NULL S)
NUMBERP	Dà T se l'argomento è un numero	(NUMBERP S)
OR	Dà il valore R se almeno una S-espressione è diversa da NIL, altrimenti dà NIL	(OR S1...Sn)
PLUS	Somma fra loro tutti gli argomenti	(PLUS X1...Xn)
PLUSP	Dà T se l'argomento è positivo	(PLUSP X)
PRINC	Come PRINT, ma PRINT dà un "ritorno carrello" prima di iniziare e uno spazio alla fine	(PRINC S)

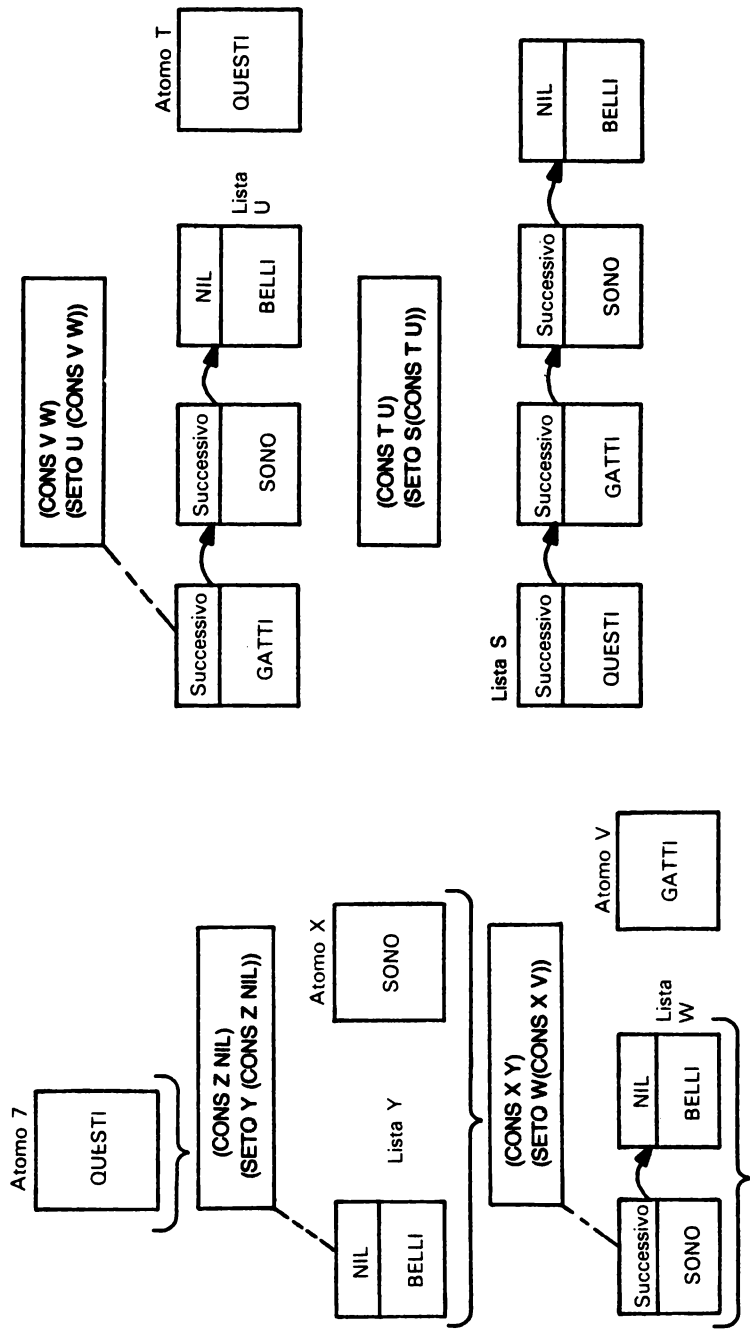



Fig. 2.9. La funzione CONS.

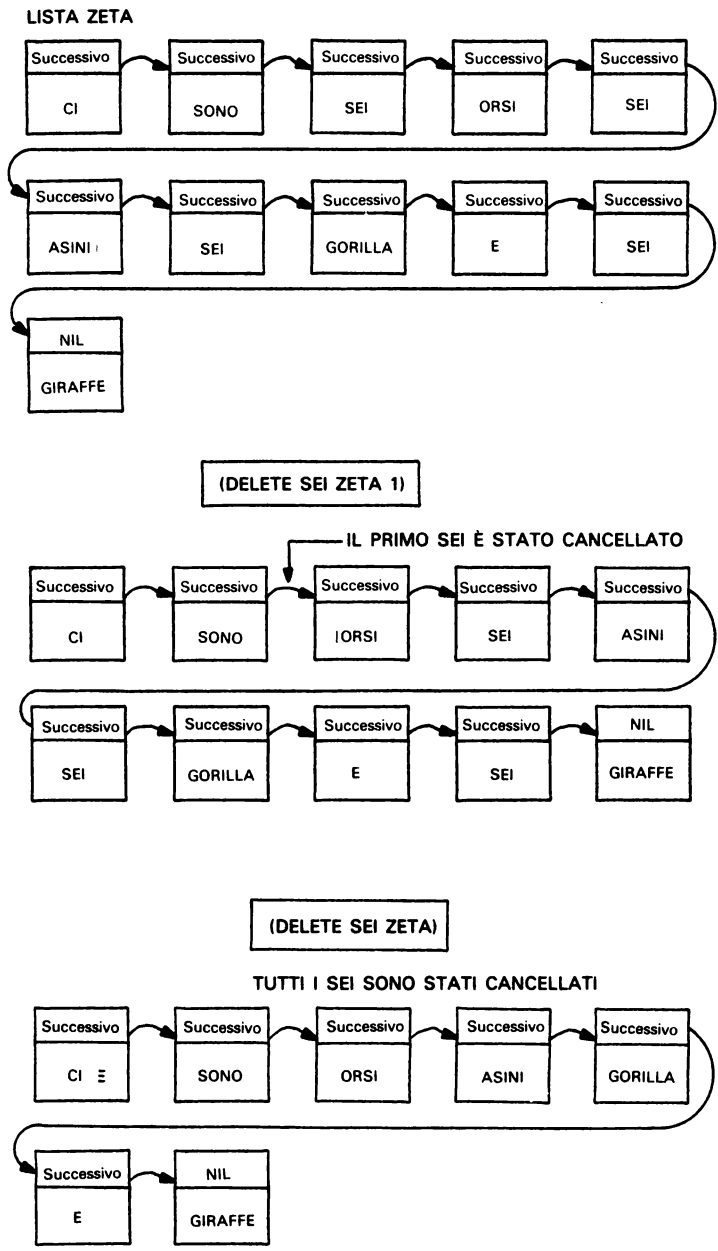



Fig. 2.10. La funzione DELETE.

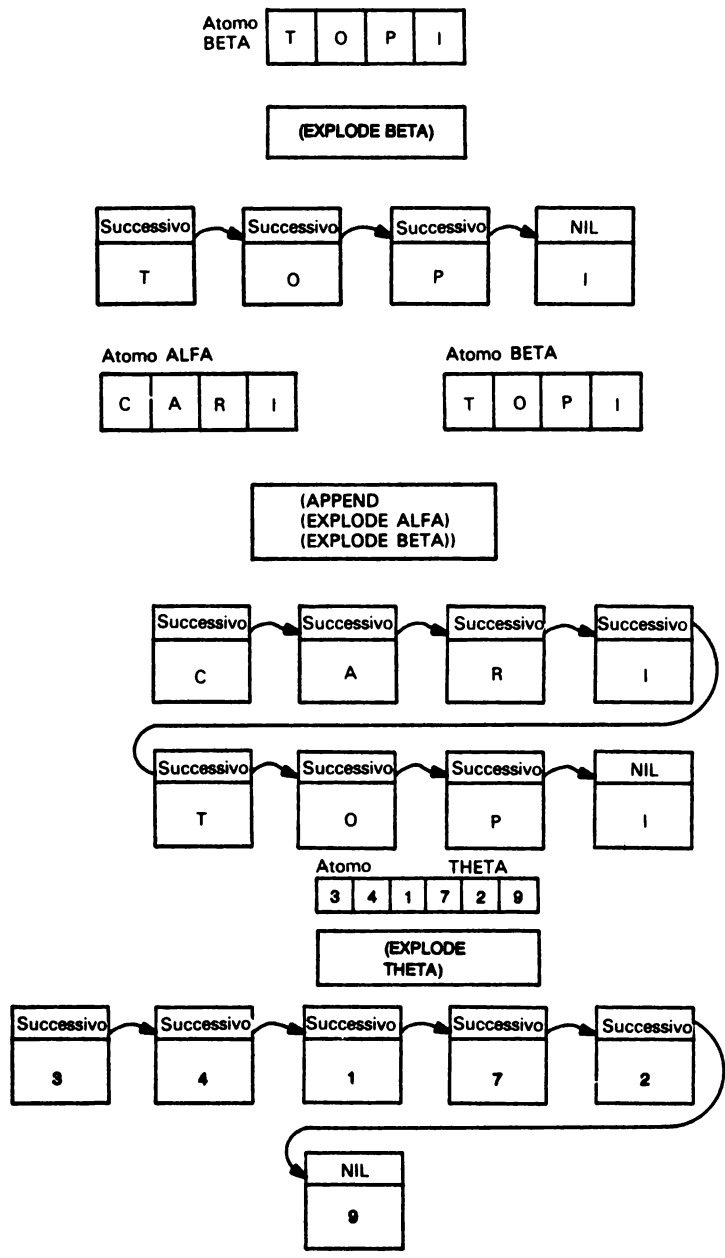



Fig. 2.11. La funzione EXPLODE.

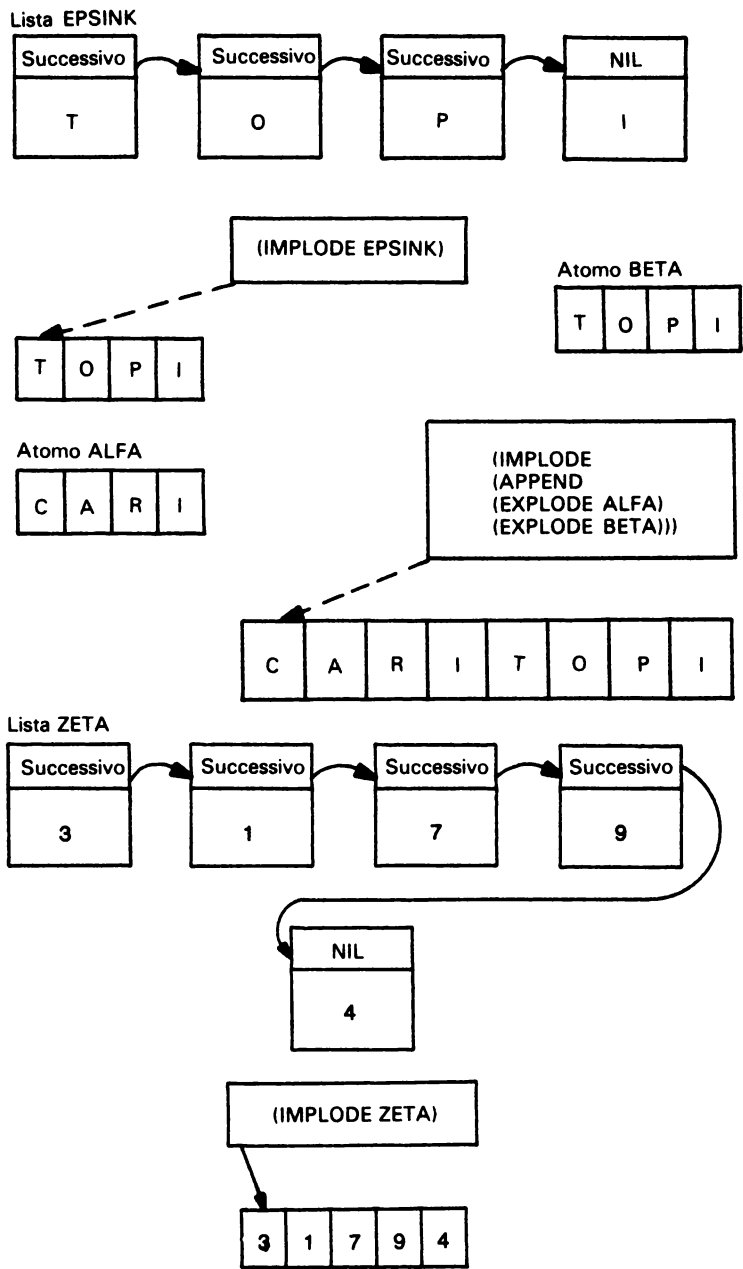



Fig. 2.12. La funzione IMPLODE.

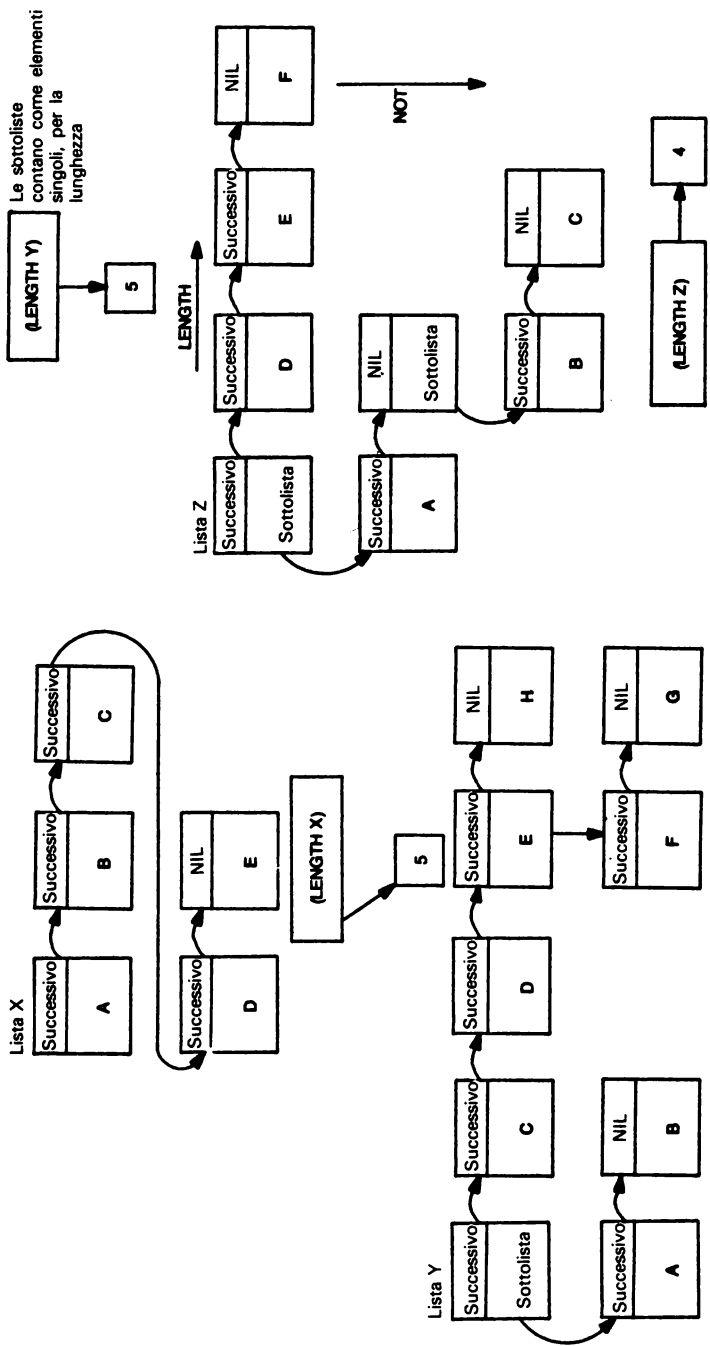



Fig. 2.13. La funzione LENGTH.

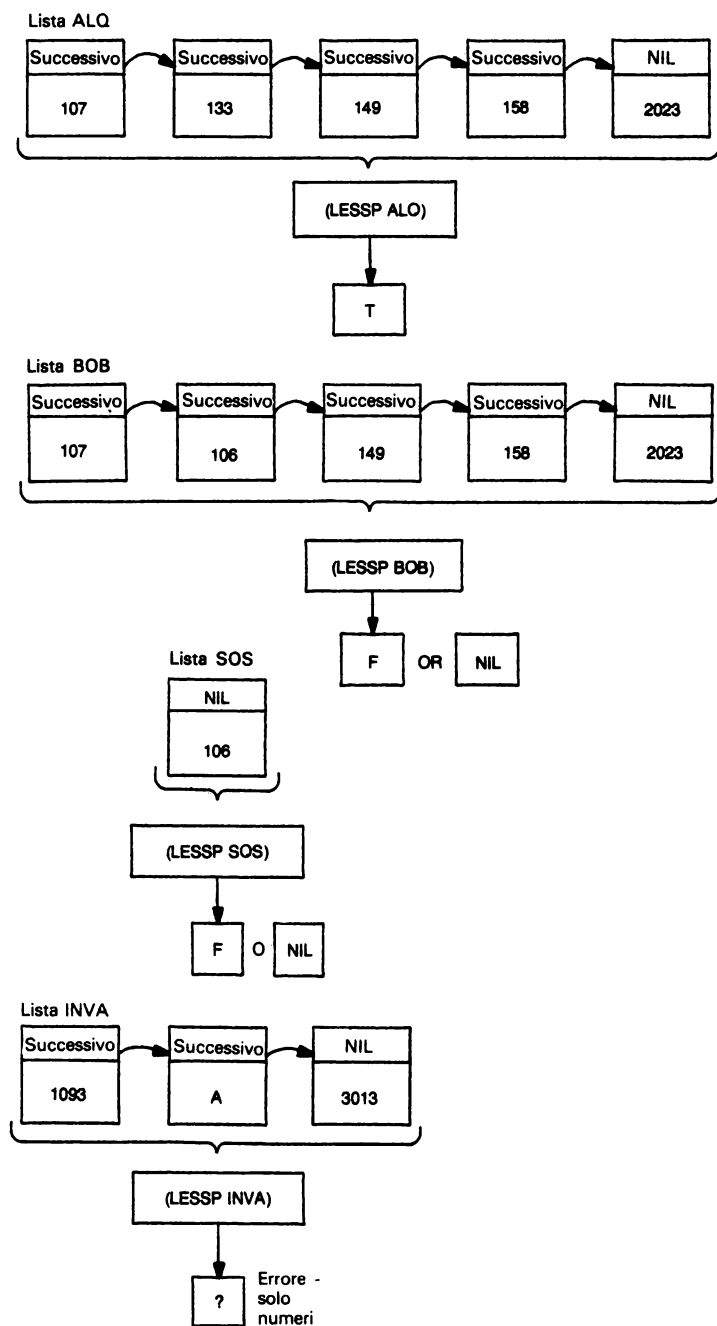



Fig. 2.14. La funzione LESSP.

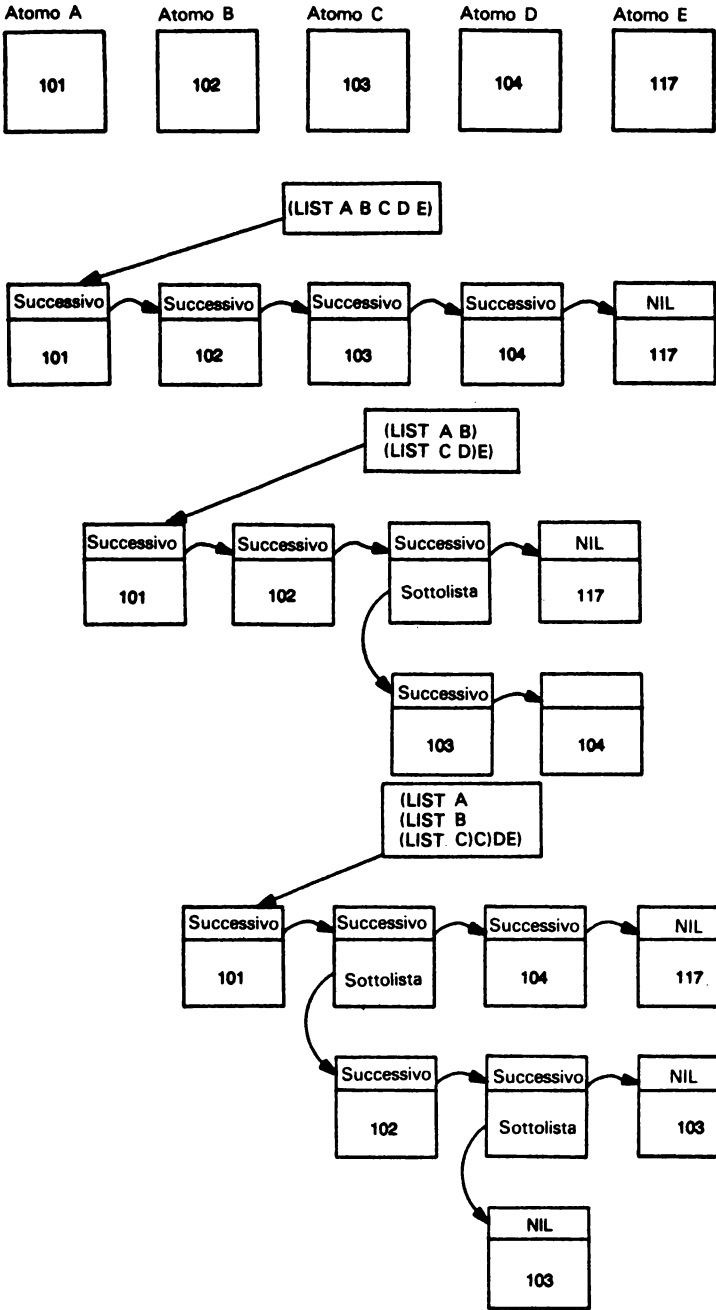



Fig. 2.15. La funzione LIST.

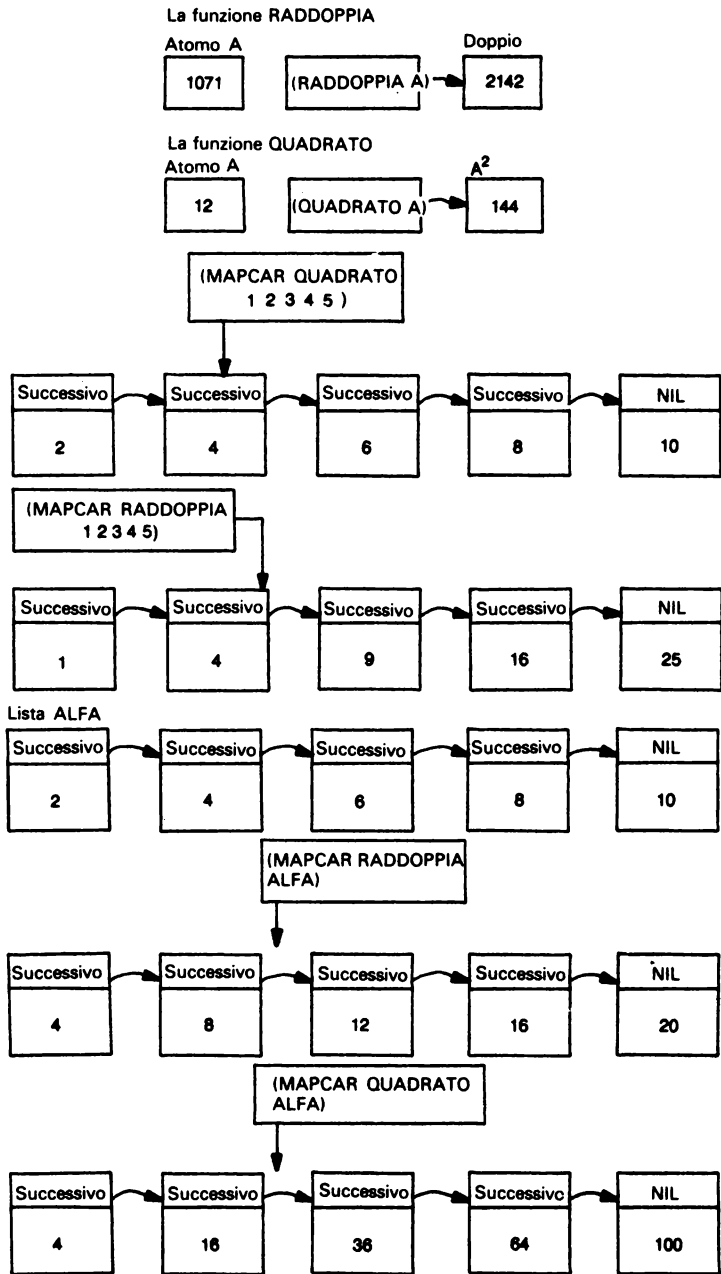



Fig. 2.16. La funzione MAPCAR.

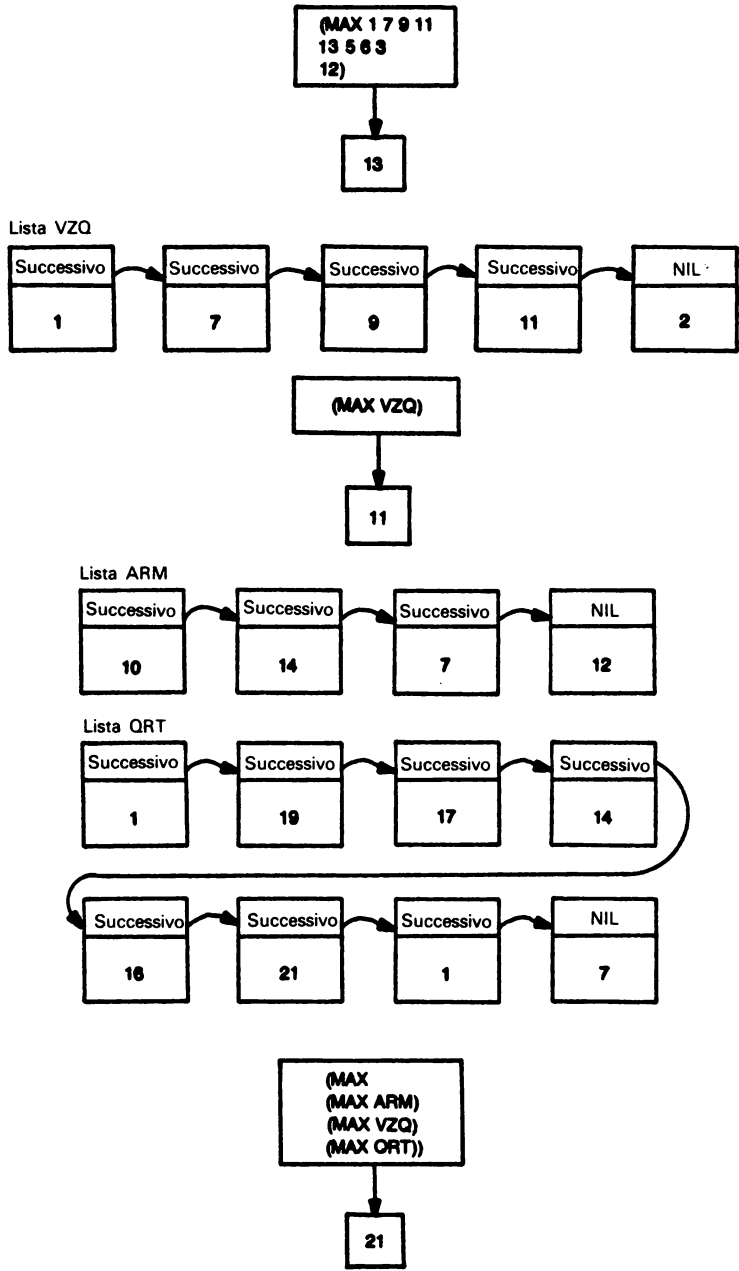



Fig. 2.17. La funzione MAX.

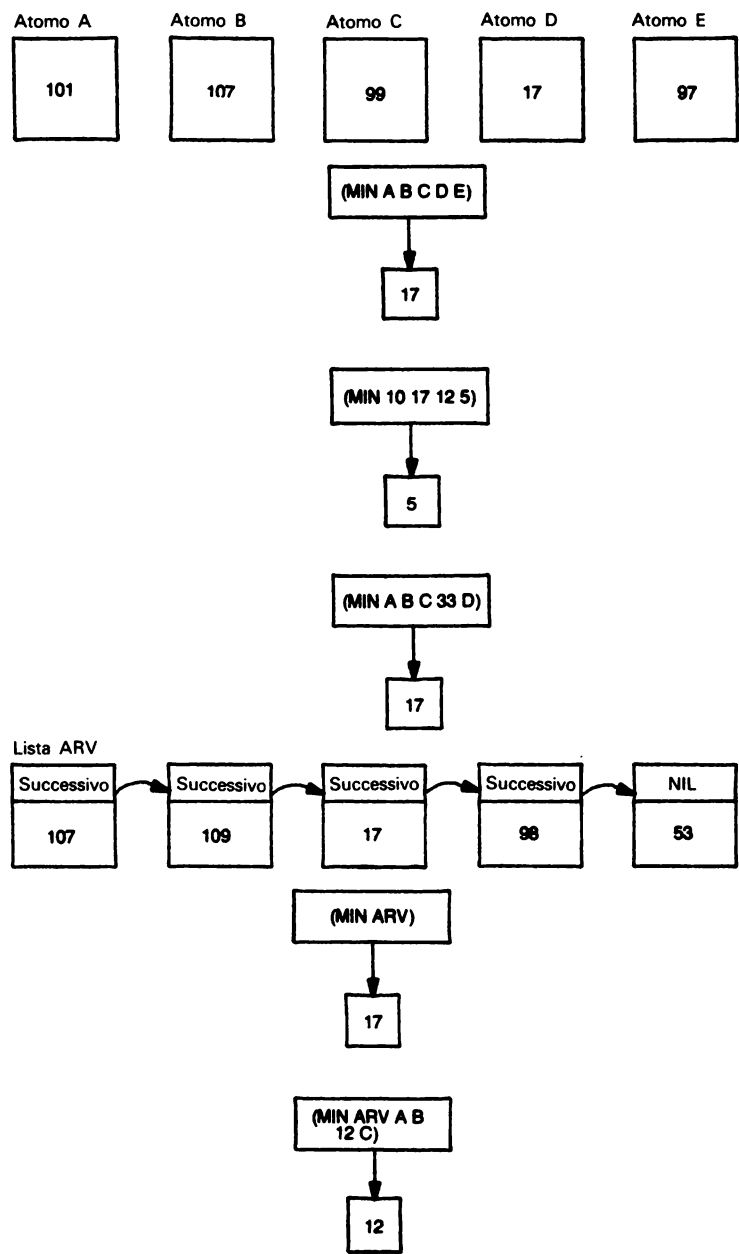



Fig. 2.18. La funzione MIN.

PRINT	Fa stampare la S-espressione; il valore è sempre T	(PRINT S)
PROG	Vedi nel testo: crea variabili e permette l'iterazione	
PUTPROP	Memorizza la S-espressione come proprietà data dell'atomo che appare come primo argomento	(PUTPROP A S)
QUOTE	Del tutto identica alla S-espressione	(QUOTE S)
QUOTIENT	Dà il risultato della divisione del primo argomento per il secondo	(QUOTIENT X X)

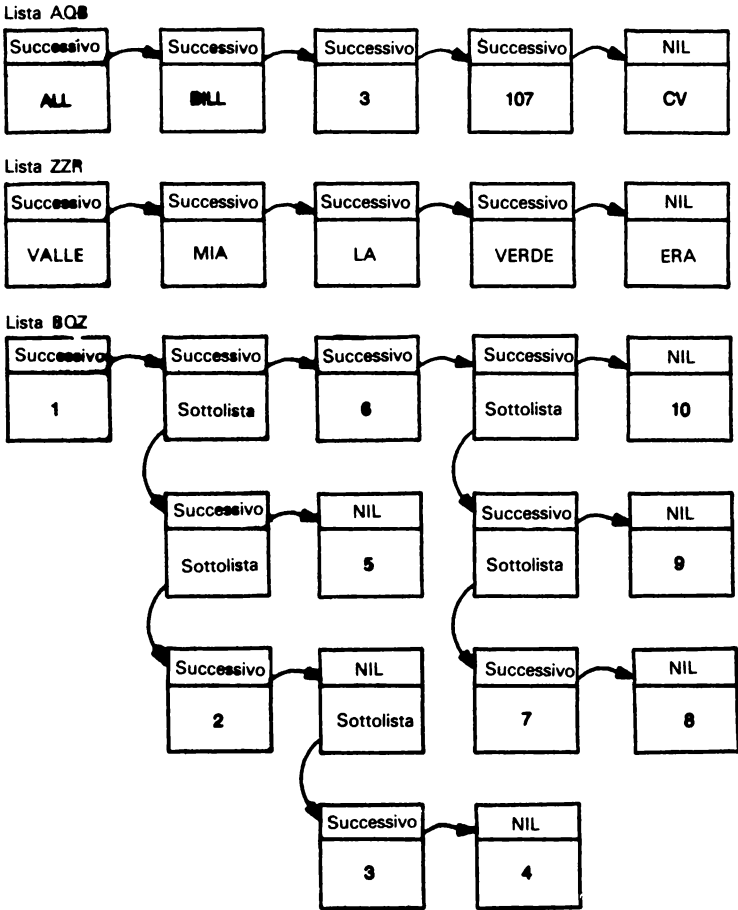



Fig. 2.19. La funzione REVERSE.

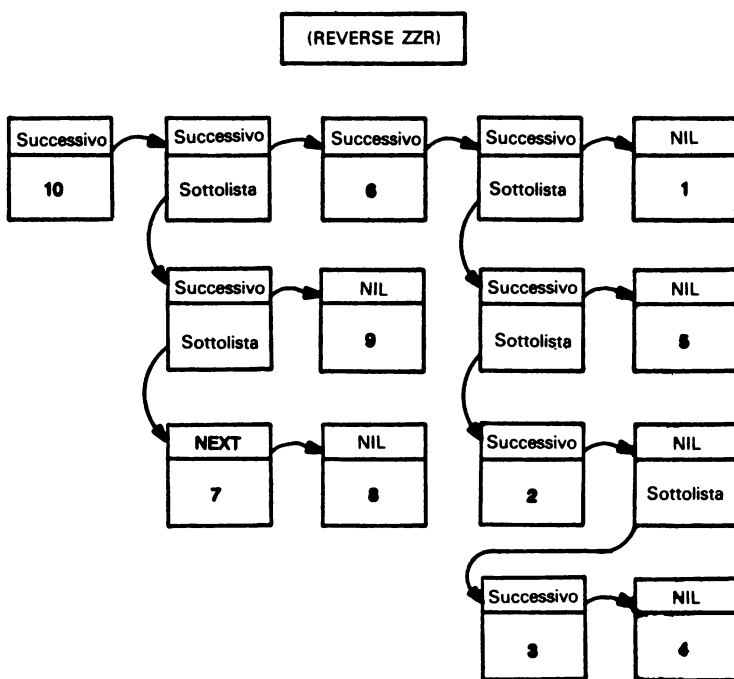
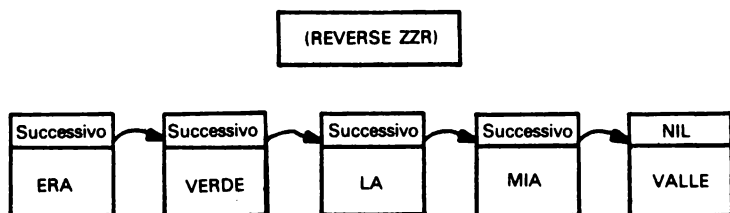
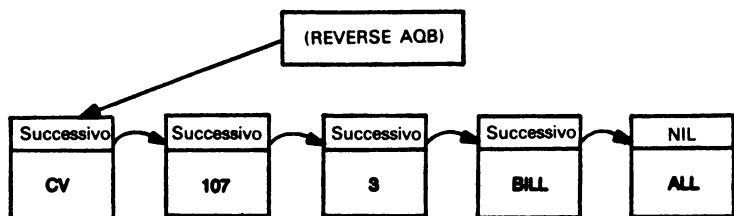



Fig. 2.19. (segue).

READ	Fà sì che la S-espressione venga data come valore del PROG in cui appare	(READ S)
REVERSE	Inverte l'ordine degli elementi in una lista	(REVERSE L) (fig. 2.19)
RETURN	Fa sì che la S-espressione venga restituita come valore del PROG in cui appare	(RETURN S)
SET	Dà il secondo argomento. Il primo argomento deve essere valutato come un atomo e il valore di quest'atomo diventa il valore del secondo argomento	(SET S1 S2)

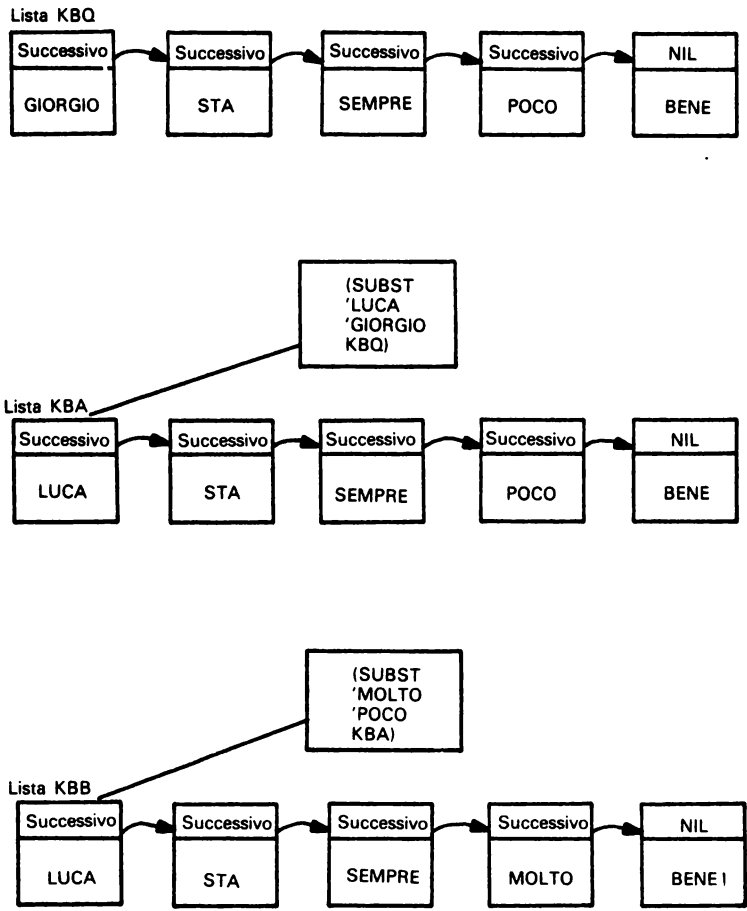



Fig. 2.20. La funzione SUBST.

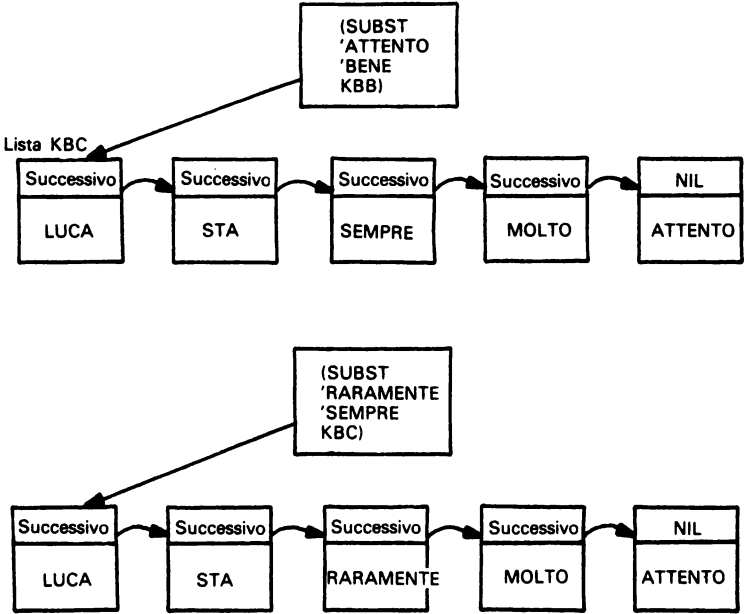



Fig. 2.20. (segue).

SETQ	Dà la S-espressione e il suo risultato è quello di rendere la S-espressione valore dell'atomo	(SETQ A S)
SQRT	Dà la radice quadrata dell'argomento	(SQRT X)
SUB1	Dà l'argomento diminuito di 1	(SUB1 X)
SUBST	Sostituisce la prima S-espressione per tutte le occorrenze della seconda S-espressione nella terza S-espressione	(SUBST S1 S2 S3) (fig. 2.20)
TERPRI	Provoca l'esecuzione di un ritorno carrello	(TERPRI)
TIMES	Dà il prodotto di tutti gli argomenti	(TIMES X1...Xn)
ZEROP	Dà il valore T se l'argomento è 0	(ZEROP X)

LOGICA

Il Lisp è un linguaggio funzionale, simbolico, ricorsivo e per l'elaborazione di liste. C'è ancora qualcosa?
Il Lisp è anche un linguaggio *logico.*

Che cosa significa il LAMBDA *quando si costruiscono nuove funzioni?*
LAMBDA definisce l'argomento usato dalla nuova funzione, per esempio:

```
(LAMBDA(X Y)(DIFFERENCE(TIMES X Y)(QUOTIENT X Y)))
```

significa che gli argomenti sono usati in questo ordine e che la funzione creata è XY−X/Y.

Che cosa sono esattamente gli argomenti di LAMBDA*?*
In generale, il primo argomento di LAMBDA è la lista degli argomenti usati dalla nuova funzione. Il secondo argomento di LAMBDA è la S-espressione che usa tali argomenti (fig. 2.21).

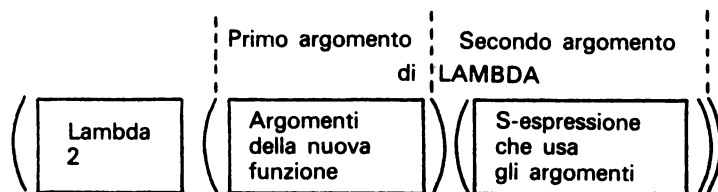



Fig. 2.21.

Che cosa succede quando si chiama LAMBDA*?*
La chiamata di LAMBDA è a sua volta una descrizione di funzione in termini di valore.

Come si immette la funzione costruita con LAMBDA?
La si immette come segue:

```
(LAMBDA(X Y)(DIFFERENCE(TIMES X Y)(QUOTIENT X Y)))(4 2)
```

Che cosa significa il (4 2) *alla fine dell'espressione con* LAMBDA*?*
Il (4 2) è la lista degli argomenti che l'espressione LAMBDA deve usare per valutare la funzione costruita (fig. 2.22).

Quali valori può avere LAMBDA*?*
In Lisp, LAMBDA non può avere un valore che sia un nome, un numero

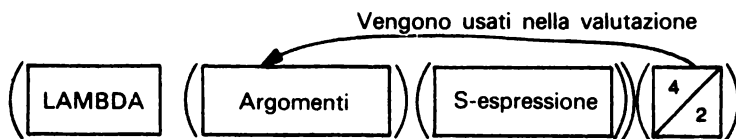



Fig. 2.22.

di una lista, ecc. In questo esempio di LAMBDA (e in generale per qualunque esempio di uso della costruzione con LAMBDA) bisogna ricordare che LAMBDA non è proprio una funzione, benché abbia argomenti come le funzioni e compaia in prima posizione nella sua lista.

La storia di LAMBDA *ha qualche significato per il suo comportamento?*
Sì, la costruzione con il LAMBDA risale al 1941: la presentò Alonzo Church nel suo libro *The Calculi of LAMBDA-Conversion*. Nella sua notazione, l'uso di LAMBDA, per il nostro esempio, sarebbe $\lambda x(xy-x/y)$. In Lisp, bisogna essere molto precisi nell'espressione. Tanto per cominciare, *xy* sarebbe un atomo a sé e non la coppia di atomi *x* e *y*. (Va notato che in Lisp si usano tipicamente le lettere maiuscole. Qui ho usato le minuscole solo per spiegare la costruzione della formula di Church.)

Come si definisce una funzione con un nome?
In primo luogo si scrive una semplice definizione con un nome, poi si fa seguire la spiegazione:

```
DEFINE (((K(LAMBDA(S)(PLUS S S)))))
```

Qui la costruzione è al livello più alto (si riveda in proposito il paragrafo sul sistema Lisp). Se si vuole scrivere questo esempio a un livello più interno (come si farebbe in un programma) si deve riportare:

```
(DEFINE((QUOTE(K(LAMBDA(S)(PLUS S S))))))
```

In ogni caso questa procedura definisce un'unica funzione data da una lista di due elementi. Il primo elemento è K, il nome della funzione; il secondo è la descrizione della funzione. Questa descrizione specifica che la funzione data ha "un solo" argomento. Nella lista che è il primo argomento di LAMBDA è dato solo l'atomo S. Questo significa che in futuro, ogniqualvolta si chiamerà la funzione K, sarà necessario un solo argomento.

Come si richiamerebbe la funzione che abbiamo appena costruito?
Siccome ora K è una istruzione (funzione) valida, la si richiama come si

farebbe con qualunque altra funzione. In termini generali, ("nome della funzione" "argomento"). In questo caso dovresti scrivere (K 2) se volessi valutare la funzione per l'argomento 2.

La nuova funzione può essere usata come qualunque altra funzione?
Sì, la si può usare esattamente come se fosse una funzione standard che risiede regolarmente nel Lisp. Si possono anche costruire nuove funzioni usando la nuova funzione (nel secondo esempio, la funzione K).

Che cosa sono i parametri "formali" ed "effettivi"?
Se in Fortran si scrive:

```
FUNCTION F(A, B, C)
```

A, B, C sono i parametri formali. Se si usa la funzione in una espressione come:

```
SDF(U/H, E-R, Z)
```

U/H, E-R e Z sono i parametri effettivi. I parametri effettivi a volte possono essere espressioni, mentre i parametri formali non sono *mai* più complessi di singoli identificatori. Tuttavia i parametri formali possono stare per matrici o funzioni oltre che per variabili. Esiste ovviamente una corrispondenza fra parametri formali ed effettivi. Nella "espressione effettiva" che usa la FUNCTION creata i parametri effettivi sono in corrispondenza biunivoca con i parametri formali. Qualunque cosa FUNCTION specifichi si debba fare con il primo parametro formale della lista viene effettuato sul primo parametro effettivo nell'espressione. Tutto quello che FUNCTION specifica si deve fare sul secondo parametro formale della definizione viene eseguito sul secondo parametro effettivo nell'espressione — e via di seguito (fig. 2.23).

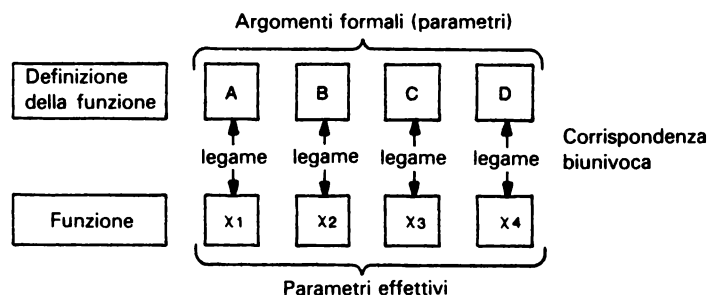



Fig. 2.23.

Che cosa significa "vincolare"?
L'idea della corrispondenza biunivoca fra parametri formali ed effettivi è l'analogo del vincolare una variabile nella logica matematica.

E questo come si applica al Lisp?
In Lisp, fra parametri formali ed effettivi si svolge lo stesso tipo di attività. Se riguardiamo la S-espressione data nel paragrafo sulla costruzione LAMBDA nella sezione sulla logica, si può vedere che X e Y sono i parametri formali e 4 e 2 sono i parametri effettivi. X è legato a 4 e Y è legato a 2 (fig. 2.24).

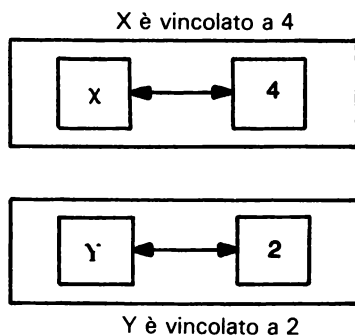



Fig. 2.24.

Questo comporta che anche ogni variabile nel secondo argomento di una espressione LAMBDA *è un parametro formale?*
Nell'esempio, le variabili che occorrono nel secondo argomento sono in effetti parametri formali. Notate che quando uso la parola "parametro" voglio dire "variabile vincolata". Nel Lisp si possono avere anche variabili "libere" nel secondo argomento di una espressione definitoria. Una variabile libera usa come suo valore il valore attuale o presente al momento dell'esecuzione dell'espressione contenente la variabile libera. Non è vincolata a valori dati con l'espressione.

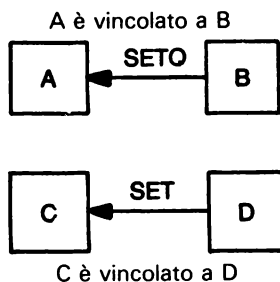



Fig. 2.25.

Che differenza c'è fra vincolare una variabile e assegnarle un valore?
Vincolare una variabile e assegnare un valore a una variabile sono operazioni che il Lisp tratta internamente più o meno allo stesso modo (fig. 2.25). Per esempio, ogniqualvolta si esegue una operazione SETQ o SET in effetti si stanno vincolando delle variabili.

Se le cose stanno così, le operazioni SETQ *e* SET *potrebbero essere chiamate operazioni vincolanti?*
Sì, e più avanti parlerò del chiamare una funzione usando SETQ o SET come di un "vincolare una variabile" (fig. 2.26). Per esempio, con

(SETQ X 6)

si vincola la variabile X a 6, e si può scrivere:

(SET(QUOTE A)(QUOTE (4 7 9)))

e dire che si vincola A al valore (4 7 9).

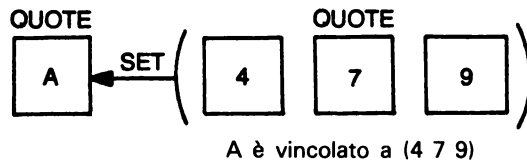



Fig. 2.26.

ELABORAZIONE DI LISTE

In che modo si può usare il Lisp ora?
Si possono effettuare chiamate di funzioni di sistema (funzioni standard o già definite in Lisp) e si possono definire funzioni semplici.

Che cosa è necessario per definire funzioni più complesse?
Sono necessarie le funzioni di elaborazione di liste del Lisp.

Che cosa sono queste funzioni di elaborazione di liste?
Per il momento, vediamo solo le funzioni più primitive, CAR, CDR e CONS. Prima di andare oltre, va notato che tutte le funzioni del Lisp vengono pronunciate come parole, e mai come sigle. La funzione CDR viene letta "*couder*" e mai pronunciata come ci-di-erre.

Perché CAR, CDR *e* CONS *sono dette funzioni primitive di lista?*

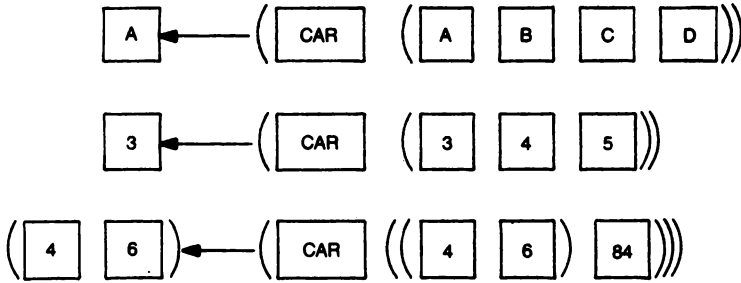



Fig. 2.27.

Perché tutte le altre funzioni di lista sono basate su queste tre.

Che cosa fa la funzione CAR?
La funzione CAR prende come primo argomento una lista e dà come valore il primo elemento della lista stessa (fig. 2.27). La lista deve avere un nome, di solito una sola lettera, e si può far riferimento alla lista usando il suo nome.
Consideriamo queste liste:

se A è (3 4 5)
se B è (5(3(6(4))))
se C è ((4 6) (84))
se D è ((((5)9)5)3),

allora

(CAR A) è 3
(CAR B) è 5
(CAR C) è (4 6)
(CAR D) è (((5)9)5)

Che cos'è la funzione CDR?

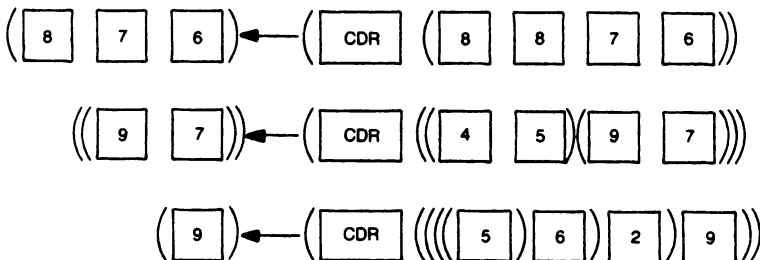



Fig. 2.28.

La funzione CDR prende come argomento (come CAR) una lista (fig. 2.28). Il suo valore è quel che resta della lista dopo aver tolto il suo primo elemento. Per esempio:

se A è (8 8 7 6)
se B è (3(7(9(3))))
se C è ((4 5)(9 7))
se D è ((((5)6)2)9)

allora

(CDR A) è (8 7 6)
(CDR B) è ((7(9(3))))
(CDR C) è ((9 7))
(CDR D) è (9)

Quand'è che (CAR A) *non ha senso?*
Se A sta per un atomo, (CAR A) non ha significato e provocherà la segnalazione di un errore di sistema.

Quand'è che (CDR A) *non ha senso?*
Se A sta per una lista che contiene esattamente un elemento, (CDR A) non ha senso, se si considera la definizione di CDR data prima. Tuttavia, nel caso di una lista che contiene solo un elemento, (CDR A) dà come risultato l'atomo speciale NIL. Che, se ben ricordate, significa anche "falso". Questa è un'altra funzione dell'atomo NIL (fig. 2.29).

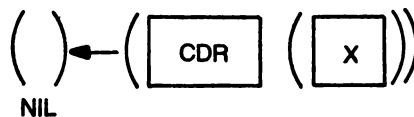



Fig. 2.29.

Abbiamo visto CAR *e* CDR. *E* CONS*?*
CONS è la funzione che rimette insieme quello che CAR e CDR hanno separato.

Come funziona CONS *nel Lisp?*
CONS ha come valore una lista. I suoi argomenti (ne richiede due) sono i risultati delle operazioni CAR e CDR, rispettivamente, applicati a tale lista (fig. 2.30). Consideriamo qualche esempio:

se A è 5 e B è (6 7 9)
se C è 9 e D è ((3(5(8))))

se E è (3 4) e F è ((3 5))
se G è (((5)8)3) e H è (3)

allora

(CONS A B) è (5 6 7 9)
(CONS C D) è (9(3(5(8))))
(CONS E F) è ((3 4)(3 5))
(CONS G H) è (((5)8)3)3)

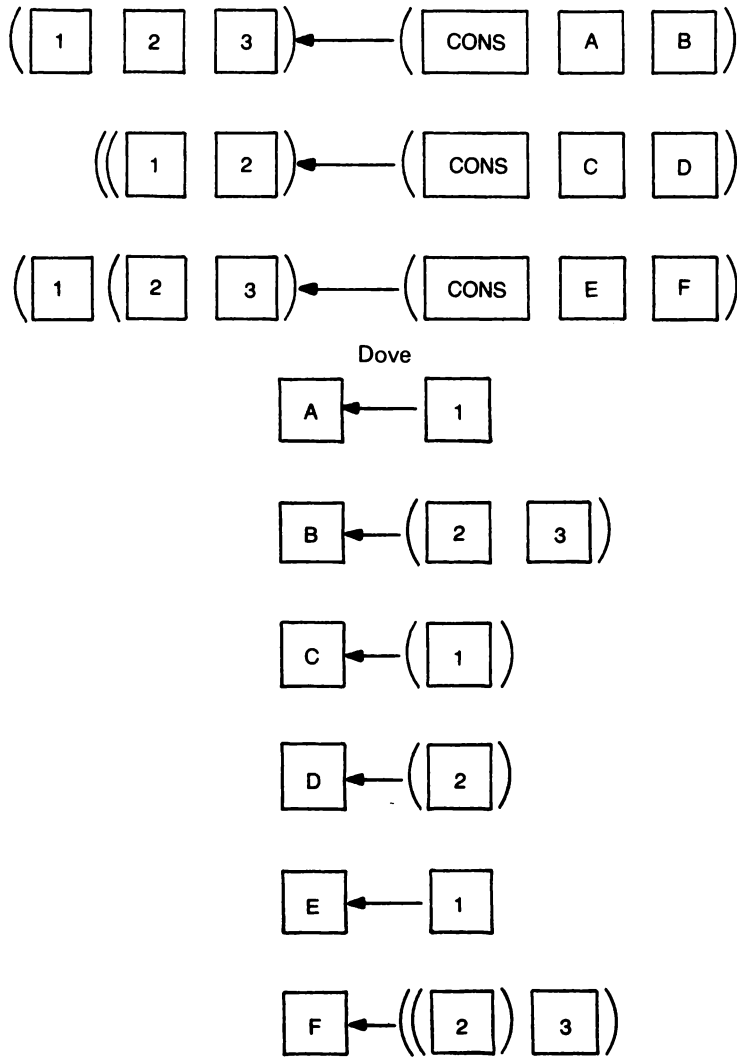



Fig. 2.30.

I risultati prodotti da CONS, applicata a due argomenti qualunque, sono i seguenti. Il secondo argomento deve essere una lista. Per ottenere il valore di CONS, si aggiunge un ulteriore elemento davanti a questa lista. Questo elemento sarebbe il primo argomento di CONS, e può essere qualunque cosa, anche un atomo.

Le definizioni di CAR, CDR *e* CONS *sono complete?*
No, ma ne parleremo ancora più avanti.

Che cos'è il secondo elemento di una lista qualunque?
Il primo elemento del CDR della lista. Se la lista è per esempio:

(6 7 8)

Ovviamente, il secondo elemento è 7 e il valore dell'operazione CDR su questa lista è (7 8). Il 7 è il primo elemento di quel valore.

In generale, se si vuole isolare il secondo elemento, che cosa si fa?
Si combinano le operazioni CAR e CDR. Ipotizziamo di avere una lista chiamata K e che tale lista sia:

(4 5 6 7)

Se si scrive:

(CAR(CDR K))

si ottiene come valore l'elemento 5. L'operazione è diretta. Il (CDR K) ha come valore (5 6 7), gli elementi che restano dopo l'eliminazione del primo elemento. Se si applica la funzione CAR alla nuova lista (5 6 7) si ottiene il valore 5. Le funzioni di lista vengono usate esattamente come le funzioni aritmetiche. Pertanto, si nota che la S-espressione più interna

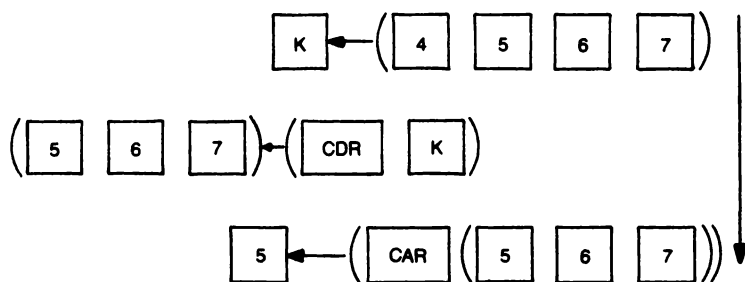



Fig. 2.31.

(CDR K) viene valutata per prima. Poi la funzione CAR usa come argomento il valore prodotto dalla prima operazione, CDR (fig. 2.31).

È possibile scrivere in un altro modo (CAR(CDR L)), *dove* L *è una lista qualunque?*
Sì, si può scrivere (CADR L), che è esattamente identico a (CAR(CDR L)).
Provate a fare qualche esperimento con (CAR(CDR L)). Usate le liste seguenti:

(2 3 4)
((5 6)(7 8)(7 6))
(5 9 3 7 6)

Ipotizzando che le liste abbiano lo stesso nome S, usate la funzione (CAR(CDR S)). Potete usare anche la funzione composta (CADR S).

(CADR S) è 3, quando S è (2 3 4)
(CADR S) è (7 8), quando S è ((5 6)(7 8)(7 6))
(CADR S) è 9, quando S è (5 9 3 7 6)

Si può estrarre con altrettanta facilità anche il terzo elemento?
Sì, si può scrivere:

(CADR(CDR L))

oppure

(CAR(CDR(CDR L)))

In ogni caso, il terzo elemento della lista L è il secondo elemento di (CDR L) (fig. 2.32).

Questa funzione può essere scritta anche in un modo più facile?
Sì, è possibile scrivere

(CAR(CDR(CDR L)))

nella forma

(CADDR L)

dove (CADDR L) ha esattamente lo stesso significato di (CAR(CDR(CDR L))) ovvero di (CADR(CDR L)).

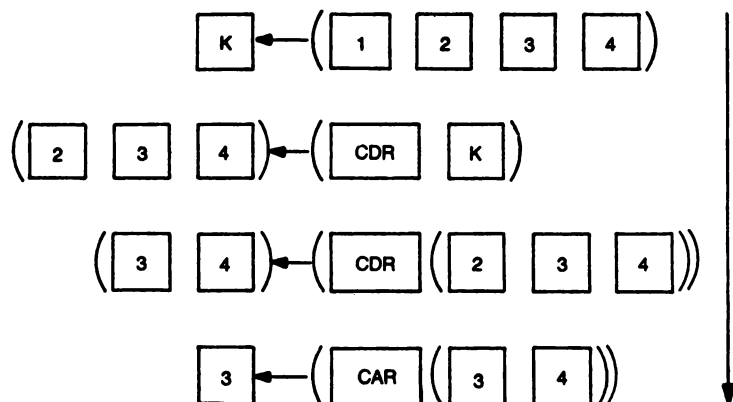



Fig. 2.32.

Esiste qualche regola generale da seguire quando si costruiscono queste funzioni di più agile scrittura?

Sì, in generale in Lisp l'utente può creare funzioni speciali per selezionare elementi usando la regola seguente. Il nome della funzione deve iniziare con una C, poi è seguito da un numero ragionevole di A e di D — di solito fino a un massimo di cinque circa. Ogni A sta per la funzione CAR, mentre ogni D sta per la funzione CDR. La funzione creata deve poi finire con una R. Pertanto, si possono creare molte funzioni diverse: nella funzione composta le A e le D sono eseguite nell'ordine da sinistra verso destra.

Il Lisp possiede qualche tipo di funzione di concatenazione?

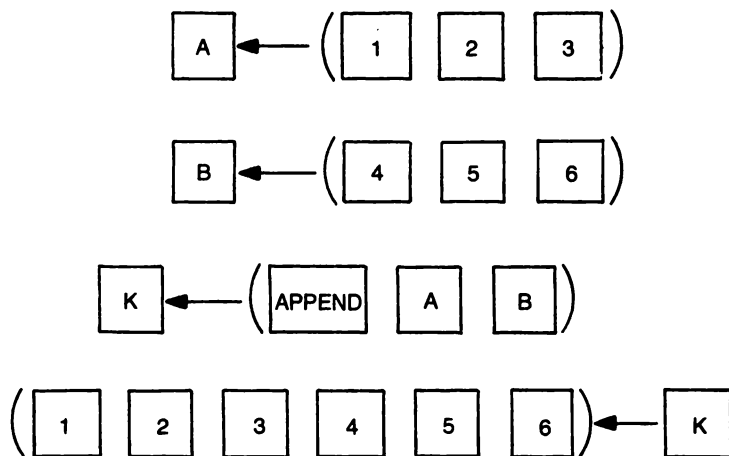



Fig. 2.33.

Sì, la funzione APPEND (fig. 2.33). Concateniamo qualche esempio:

se A è (3 4 5) e B è (6 7)
se C è (4 5 6 7 8) e D è (1 2 3)
se E è (1 1 1) e F è (2 2 2)

allora

(APPEND A B) è (3 4 5 6 7)
(APPEND C D) è (4 5 6 7 8 1 2 3)
(APPEND E F) è (1 1 1 2 2 2)

Notate poi che se si scrivesse:

(CONS E F)

si otterrebbe come risultato:

(CONS E F) è ((1 1 1)2 2 2)

PREDICATI

Una frase in italiano di quali parti si compone?
Se è una normale frase dichiarativa, possiede un soggetto e un predicato (fig. 2.34).

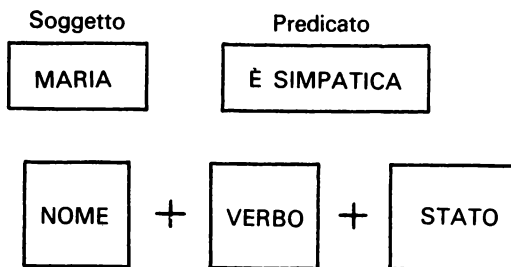



Fig. 2.34.

Che cosa ha a che fare questo con il Lisp?
I predicati possono essere veri o falsi. Pensate a qualche esempio di frase in italiano, per cominciare. "Lisp è un linguaggio per calcolatori." "Un atomo è costituito da due S-espressioni sommate." "La funzione

CAR dà il primo elemento di una lista." "La funzione CDR dà l'ultimo elemento in una lista."
Nel primo enunciato, Lisp è il soggetto e "è un linguaggio per calcolatori" è il predicato. L'enunciato è "vero". Nel secondo enunciato il soggetto è "un atomo" e l'enunciato è "falso". "La funzione CAR" è il soggetto del terzo enunciato, che è "vero". "La funzione CDR" è il soggetto del quarto enunciato, che è "falso".

È una cosa importante?
I predicati sono comuni nei linguaggi di programmazione, anche se non sono chiamati predicati. Le operazioni EQ e NE in Fortran sono predicati perché ci dicono qualcosa sul soggetto e stanno rispettivamente per "uguale" e "non uguale". Il Lisp ha predicati, e anche i suoi predicati possono essere veri o falsi.

È questo ciò a cui si riferiscono T *e* F*?*
Sì, se un enunciato (un predicato) è vero, il valore del predicato è T ("*true*", vero). Se è falso ha il valore NIL. Ricordate che di solito si usa il simbolo NIL per "falso", e non F.

Un predicato è una funzione?
Tutte le costruzioni nel Lisp sono funzioni!

Che cosa fa la funzione ATOM*?*
La funzione ATOM ha come argomenti atomi o liste. Se l'argomento è un atomo, il suo valore è T (vero). Se l'argomento è una lista, il valore è NIL (falso). Si può dire che la funzione ATOM formuli la domanda "il mio argomento è un atomo?".

Come si usa la funzione ATOM*?*
Scrivete qualche enunciato e guardate. Usate un nome o un simbolo per rappresentare la lista o l'atomo:

se A è 7
se B è A (il simbolo, non preoccupatevi del valore)
se C è (4 8 2)

allora

(ATOM A) è T
(ATOM B) è T
(ATOM C) è NIL.

Questo perché in effetti A e B sono atomi, mentre C, cioé (4 8 2), è una lista.

Che cosa fa la funzione NULL*?*
La funzione NULL controlla se il suo argomento è NIL o meno. Se l'argomento è NIL, il valore di NULL è T, in ogni altro caso il suo valore è NIL.

Qual è il significato di T *nella domanda precedente?*
Sta per qualunque cosa non sia NIL, per esempio una lista o un atomo. T non è una costruzione del Lisp, ma sta semplicemente per qualunque cosa vera, cioé per qualunque cosa non sia NIL.

Qualche esempio della funzione NULL*?*
Prendiamo questi esempi:

se A è 7
se B è A (il simbolo)
se C è (4 8 2)
se D è NIL

allora

(NULL A) è NIL
(NULL B) è NIL
(NULL C) è NIL
(NULL D) è T

Ovviamente, gli argomenti 7, A, (4 8 2) non sono NIL; pertanto la funzione NULL dà un NIL. L'ultimo argomento invece è proprio NIL e pertanto la funzione NULL dà come valore T.

Che valore avrebbe NULL *in questo esempio, dove* X *è un elemento singolo:*

(NULL(CDR X))*?*

Rivedete la domanda "In quali casi (CDR A) non ha senso?" nel paragrafo sull'elaborazione di liste. Il valore di NULL sarebbe T, perché CDR dà come valore NIL, ogniqualvolta ha un argomento che è un elemento singolo.

Che cos'è la funzione EQUAL *e che cosa fa?*
La funzione EQUAL controlla se i suoi due argomenti sono uguali, o, per essere precisi, "identici". Qualche esempio:

se A è (4 5 6) e B è (4 5 6)

se C è 9 e D è 19
se E è (TIMES 5 4) e F è 20
se G è (TIMES 5 4) e H è (TIMES 5 4)
se I è ((3 4)7) e J è (3(4 7))

allora

(EQUAL A B) è T
(EQUAL C D) è NIL
(EQUAL E F) è NIL
(EQUAL G H) è T
(EQUAL I J) è NIL

Perché EQUAL dia come risultato T non è sufficiente che i valori degli argomenti siano uguali, come nel caso di E e F. Perché il risultato sia T, gli argomenti devono essere proprio identici. Esiste anche una funzione EQ che agisce solo su atomi.

Che cos'è la funzione ZEROP?
La funzione ZEROP controlla se il suo argomento è zero. La lettera P in ZEROP sta per "predicato". Se l'argomento è zero, allora il valore di ZEROP è T, altrimenti è NIL; qualche esempio:

se A è 987
se B è 56
se C è O
se D è 28

allora

(ZEROP A) è NIL
(ZEROP B) è NIL
(ZEROP C) è T
(ZEROP D) è NIL

Che cosa fa la funzione GREATERP*?*
La funzione GREATERP controlla se il suo primo argomento (è una funzione a due argomenti) è maggiore del secondo. Ecco qualche esempio:

se A è 5 e B è 7
se C è 45 e D è 44
se E è 34 e F è 81
se G è 50 e H è 50

allora

(GREATERP A B) è NIL
(GREATERP C D) è T
(GREATERP E F) è NIL
(GREATERP G H) è NIL

RIPASSO

Che cos'é (CAR A), *se* A *è* (A B C)*?*
A, perché è il primo elemento della lista, non vuota.

Che cos'è (CAR A), *se* A *è* ((SDF) H J K)*?*
((S D F)), perché è la prima S-espressione.

Che cos'è (CAR A), *se* A *è* FORMAGGIO*?*
Non c'è risposta, non si può chiedere il CAR di un atomo.

Che cos'è (CAR A), *se* A *è* 0*?*
Non c'è risposta, non si può chiedere il CAR di una lista vuota.

Per che cosa è definita la funzione CAR*?*
La funzione CAR è definita solo per liste non vuote.

Che cos'è (CAR A), *se* A *è* ((ALDO)(STA)(BENE))*?*
((ALDO))

Che cos'è (CAR A), *se* A *è* (ALDO STA BENE)*?*
((ALDO))

Che cos'è (CDR B), *se* B *è* (6 8 9)*?*
(8 9)

Che cos'è (CDR B), *se* B *è* ((S D F) J K L)*?*
(J K L)

Che cos'è (CDR B), *se* B *è* (ALDO STA BENE)*?*
(STA BENE)

Che cos'è (CDR B), *se* B *è* ((ALDO)(STA)(BENE))*?*
((STA)(BENE))

Che cos'è (CDR B), *se* B *è* (ALDO)*?*
NIL

Che cos'è (CDR B), *se* B *è* ALDO*?*
Non c'è risposta, non si può chiedere il CDR di un atomo.

Che cos'è (CDR B), *se* B *è* 0*?*
Non c'è risposta, non si può chiedere il CDR di una lista vuota.

Per che cosa è definita la funzione CDR*?*
La funzione CDR è definita solo per liste non vuote.

Che cos'è (CAR(CDR A)) *se* A *è* ((D)(T Y)(C))*?*
((T Y))

Che cos'è (CDR(CDR A)), *se* A *è* ((D)(T Y)(C))*?*
((C))

Che cos'è il CONS *dell'atomo* ALDO *e della lista* (STA BENE)*?*
(ALDO STA BENE)

Che cos'è (CONS A B), *dove* A *è* (ALDO) *e* B *è* (STA BENE)*?*
((ALDO) STA BENE)

Che cosa sono gli argomenti della funzione CONS*?*
Il primo argomento può essere un qualsiasi atomo o una qualsiasi lista; il secondo argomento deve essere una lista.

Che cos'è (CONS A B), *se* A *è* K *e* B *è* ( )*?*
(K)

Che cos'è (CONS A B), *se* A *è* ALDO *e* B *è* STA BENE*?*
Non c'è risposta, il secondo argomento deve essere una lista.

Che cos'è (CONS A B), *se* A *è* (ALDO) *e* B *è* STA BENE*?*
Non c'è risposta, il secondo argomento (qualunque cosa sia il primo) deve essere una lista.

Che cos'è (CONS A(CAR B)), *se* A *è* F *e* B *è* (C D)*?*
Non c'è risposta, perché la funzione CAR dà un atomo in questo caso e il secondo argomento di CONS deve essere una lista.

Che cos'è (CONS A(CAR B)), *se* A *è* F *e* B *è* ((C)D)*?*
(F C)

Che cos'è (CONS A(CDR B)), *se* A *è* D *e* B *è* (B C D)*?*
(D C D)

CAPITOLO TRE

Ricorsività

COME SI USANO I PREDICATI

In che cosa differiscono gli enunciati condizionali del Fortran o del Basic e quelli del Lisp?
Un enunciato condizionale in Basic o in Fortran è un enunciato del tipo "se ... allora", mentre in Lisp gli enunciati condizionali sono composti da funzioni. Bisogna ricordare che tutte le costruzioni del Lisp sono sempre composte da funzioni. Il Lisp è un linguaggio funzionale.

Qual è il nome della funzione che consente di effettuare test condizionali?
È la funzione COND.

Per esempio, come si interpreterebbe l'espressione seguente:

(COND((ZEROP A)(SETQ K 19)))*?*

Questa espressione significa "se A è zero, allora dai a K il valore 19" (fig. 3.1).

Come agisce COND*?*
La funzione chiamata COND ha un numero indefinito di argomenti. Ciascun argomento deve essere una coppia o una lista di due elementi. Il

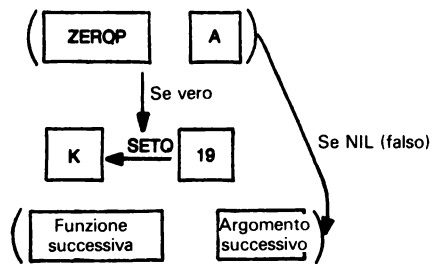



Fig. 3.1.

primo elemento della coppia è la condizione, il secondo è un'azione da svolgere o una quantità (fig. 3.2).

Che cosa significa l'espressione seguente:

(COND((ZEROP A)9)((GREATERP A C)8)(T 0))*?*

La prima funzione COND stabilisce che l'espressione è un'espressione condizionale. La prima coppia è:

((ZEROP A)9)

e, se A è 0, l'espressione ha valore 9. La funzione COND considera ogni

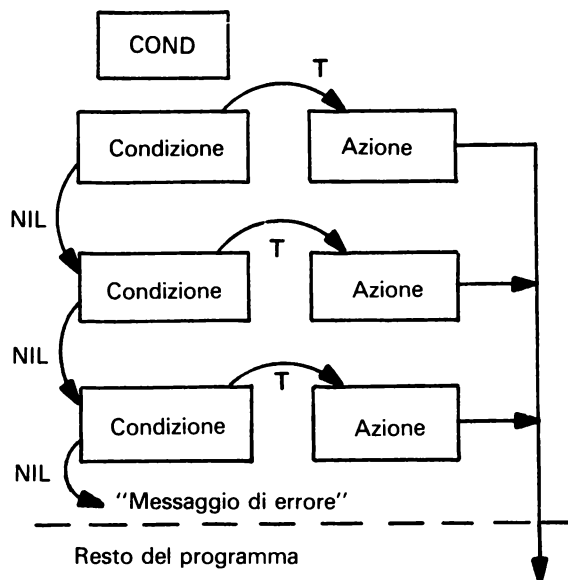



Fig. 3.2.

coppia singolarmente, procedendo da sinistra verso destra. Continua a considerare le coppie finché un test non dà valore vero, cioé finché un test non viene soddisfatto. La funzione COND non controlla se una condizione è vera, ma se è NIL. Se la condizione deve essere soddisfatta, il suo valore deve essere qualcosa di diverso da NIL. Se

((ZEROP A)9)

è soddisfatta (in altre parole, se A è 0) allora il valore dell'espressione "totale" è 9. Non appena una delle coppie è soddisfatta la funzione COND si ferma. Se

((ZEROP A)9)

non è soddisfatta, COND prende in considerazione la coppia successiva:

((GREATERP A C)8)

Se questa coppia è soddisfatta, allora il valore totale della espressione contenente la funzione COND è 8 (ipotizzando che A sia maggiore di C). Se invece

((GREATERP A C)8)

non è soddisfatta, la funzione COND prende in considerazione l'ultima

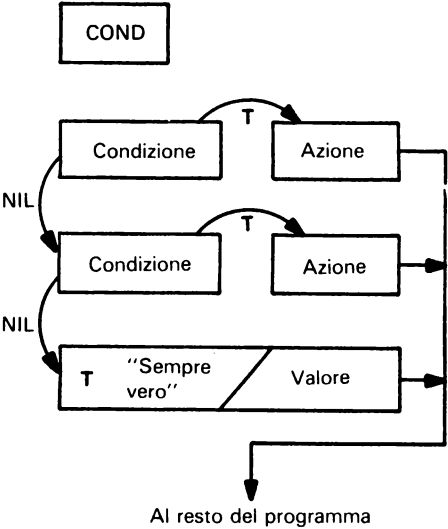



Fig. 3.3. Se non si sa per certo che almeno una condizione sia vera, usando "T" ci si assicura che almeno un predicato sia vero, cosicché non verrà generato un messaggio di errore.

coppia:

(T 0)

Poiché T è sempre diverso da NIL, la coppia è soddisfatta e il valore dell'espressione è 0.

L'ultima coppia di una espressione contenente la funzione COND *deve essere sempre del tipo* (T x), *dove* x *può essere una cosa qualunque?*
Sì, perché se nessuna condizione fosse soddisfatta, l'espressione condizionale non avrebbe un valore dotato di significato. Pertanto l'ultima coppia (T x) è una coppia "di difetto" (*default pair*): se nessun'altra coppia è soddisfatta questa determina il valore dell'espressione.

Che cosa succede se non è stata inclusa la coppia di difetto e nessuna delle coppie condizionali è soddisfatta?
Il Lisp risponde con un messaggio di errore.

FUNZIONI RICORSIVE

Che cosa è importante ricordare a proposito della scrittura di funzioni?
Prima di definire le procedure di scrittura, consideriamo ancora un attimo le parentesi. Il conto delle parentesi deve essere sempre pari, indipendentemente dal tipo di funzione che si scrive. Se non c'è il "pareggio" delle parentesi, l'espressione non verrà interpretata correttamente dall'interprete Lisp. Scoprirete che, quando si scrivono espressioni complesse in Lisp, molte espressioni si concludono con un gran numero di parentesi chiuse. È normale, perché debbono venire chiuse tutte le S-espressioni aperte durante la scrittura dell'espressione complessiva. I programmatori scelgono metodi diversi per controllare se hanno incluso nell'espressione il giusto numero di parentesi. Un metodo consiste nel contare tutte le parentesi aperte e poi contare tutte le parentesi chiuse: perché l'espressione sia scritta correttamente, il numero delle parentesi aperte deve essere uguale al numero delle parentesi chiuse. Un altro metodo sta nel considerare ciascuna sottoespressione singolarmente e nel chiedersi se quella sottoespressione è stata chiusa. Ricordate sempre che per ogni parentesi aperta deve esserci una parentesi chiusa, altrimenti il Lisp fornirà risultati errati, o non darà alcun risultato (fig. 3.4). Considerate questo esempio e provate ad applicare i due metodi da voi:

```
(DEFINE(( (FATT (LAMBDA (N)(COND((ZEROP N)1)
(T (TIMES N (FATT (DIFFERENCE N 1)))))))))
```

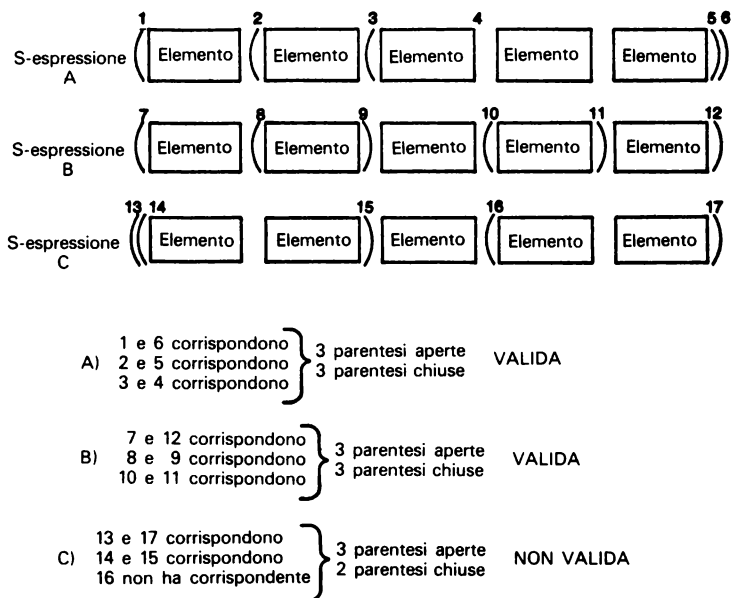



Fig. 3.4.

È facile osservare che ci sono 12 parentesi aperte e 12 parentesi chiuse. Questa espressione è la funzione per il fattoriale di un numero positivo. La parola FATT sta per il nome della funzione che viene definita. L'espressione LAMBDA dichiara quanti argomenti richieda la funzione FATT.

Le parentesi sono molto importanti in Lisp. La parola DEFINE è al livello 0 e deve essere seguita da tre parentesi. Separate le prime due così che, se viene definita più di una funzione, si può pensare a ciascuna funzione ulteriore come preceduta da una parentesi aperta. Tipicamente, le funzioni iniziano con un nome di funzione, seguito da LAMBDA e da una lista di argomenti formali, poi ancora da COND (le condizioni della funzione).

Che cos'altro bisogna notare?

Se la S-espressione è molto lunga, può essere scritta su più righe. Il Lisp è un linguaggio a formato libero. Se preferite, potete dire che i programmi in Lisp sono scritti in un formato a campo libero. Il Lisp riconosce le S-espressioni in cui le parentesi sono in numero pari e non bada a che cosa figura su una data riga. Si possono scrivere più enunciati per riga o un solo enunciato per riga. Mentre imparate il Lisp, limitate il numero degli enunciati per ciascuna riga. La lettura del programma è più facile se ogni riga contiene tutti gli elementi essenziali della sottoespressione scritta su quella riga (fig. 3.5).

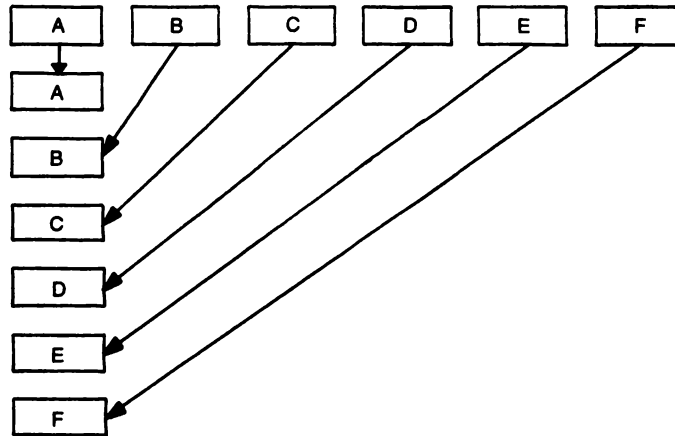



Fig. 3.5. Questa impostazione, con un enunciato per riga, aumenta la leggibilità del programma e l'affidabilità del software. Riuscire a vedere ogni passo singolarmente, anziché solo il tutto, aumenta notevolmente la possibilità di mettere a punto i programmi, in particolare quando sono lunghi o di natura complessa. Inoltre, presentando il programma al sistema enunciato per enunciato assicura che nella memoria di transito del terminale non si trovi materiale erroneo.

ELABORAZIONE RICORSIVA DI LISTE

Quali sono le funzioni fondamentali per l'elaborazione di liste?
Sono CONS, CAR e CDR.

Quando si scrivono funzioni ricorsive per l'elaborazione di liste, quali sono le funzioni (standard) usate più spesso?
Le più usate, nella definizione di nuove funzioni che siano funzioni ricorsive di elaborazione di liste, sono CAR e CDR.

Come si "crea" la funzione ADD?
Si scrive l'espressione seguente:

```
DEFINE(( (ADD (LAMBDA (K) (COND
((NULL K) 0)(T(PLUS(CAR K)(ADD(CDR K)))) )))))
```

Che cosa fa la funzione ADD?
La funzione ADD somma gli elementi in una lista chiamata K.

Qualche esempio della funzione ADD?

Se K è (1 2 3 4) allora (ADD K) è 10;
se K è (34 56 78 90) allora (ADD K) è 258;

se K è (3 456 7 890 5 678 89 46 4567) allora (ADD K) è 6762;
se K è (34 35 37 38 39804 65) allora (ADD K) è 40049.

Si può fare la stessa cosa per la moltiplicazione?
Sì, si può scrivere questa espressione definitoria:

```
DEFINE(( (MULT(LAMBDA (V)(COND
((NULL V)
1)(T(TIMES(CAR V)(MULT(CDR V)))) )))))
```

Che cosa fa la funzione MULT?
La funzione MULT trova il prodotto dei numeri in una lista nello stesso modo in cui la funzione ADD trovava la somma dei numeri in una lista. Ecco qualche esempio:

se V è (2 4 6 8), allora (MULT V) è 384
se V è (67 4 10 8), allora (MULT V) è 21440.

Che cosa succede se si applica la funzione ADD *a una lista che contiene sottoliste?*
Possono succedere varie cose. In primo luogo si ottengono inevitabilmente risultati sbagliati. In secondo luogo, si può ricevere anche un messaggio di errore del Lisp. Per il modo in cui la funzione è definita, se si tenta di applicarla, prima o poi capita di voler sommare una lista a un numero, ma la funzione PLUS, usata nella definizione di ADD, non lo consente.

Si può scrivere una funzione che consenta di sommare i numeri in una lista, eventualmente anche quando vi sono sottoliste?
Sì, si può scrivere una funzione ricorsiva, chiamiamola ADL, che forma la somma dei numeri in una lista, sia che questa abbia sottoliste, sia che non ne abbia. La scriviamo così:

```
DEFINE(( (ADL(LAMBDA (G)
(COND ((NULL G)
0)
((ATOM (CAR G)) (PLUS (CAR G)
(ADL (DR G))))
(T (PLUS (ADL (CAR G))
(ADL (CDR G)))) )))))
```

Qualche esempio?

Se G è (3(4 5 6(7))), allora (ADL G) è 25;
se G è (3 3 3 (4) 3), allora (ADL G) è 16;
se G è ((6)(6)(6 6)), allora (ADL G) è 24.

Un predicato può essere definito ricorsivamente?
Sì, come ogni altra funzione in Lisp, un predicato può essere definito ricorsivamente.

Che cos'è la funzione MEMBER*?*
La funzione MEMBER, che è una funzione standard del Lisp, controlla se il suo primo argomento è presente nel suo secondo argomento. In caso positivo, dà valore T, in caso negativo dà valore NIL.

Si può avere qualche esempio del funzionamento di MEMBER*?*
Consideriamo questi casi:

se A è 6 e B è (4 5 6 7)
se C è 3 e D è (8 6 5 7 8)
se E è 1 e F è (6 5 4 1 9)
se G è 5 e G è (7 6 4 8)

allora

(MEMBER A B) è T
(MEMBER C D) è NIL
(MEMBER E F) è T
(MEMBER G H) è NIL.

Se MEMBER *non fosse una funzione standard del Lisp, come si potrebbe scriverla?*
La funzione MEMBER potrebbe essere scritta come:

```
DEFINE(( (MEMBER (LAMBDA(A B)
(COND ((NULL
B (NIL) ((EQUAL A
(CAR B)) T)
(T (MEMBER A
(CDR B)))) ))))).
```

Si possono definire più funzioni simultaneamente?
Sì, ne abbiamo già parlato. Proviamo a definire le funzioni ADL, MULT e MEMBER con una sola espressione definitoria:

```
DEFINE ((
(ADL (LAMBDA (A)
(COND(NULL A) O)
(T PLUS (CAR A) (ADL
(CDR A)))) )))
(MULT(LAMBDA (B)
(COND ((NULL B)1
```

```
T(TIMES (CAR B)
(MULT (CDR B)))) )))
(MEMBER (LAMBDA (C D)
(COND((NULL D)NIL)
((EQUAL C (CAR D))
T)(T(MEMBER C
(CDR D))) ))))).
```

LOGICA

Che cosa sono gli operatori logici in Lisp?
Sono gli operatori AND, OR e NOT. Sono molto simili ai corrispettivi in Fortran e in Basic.

Che cos'è l'operatore NOT*?*
L'operatore NOT ha un solo argomento e ha valore T se il suo argomento è NIL, mentre ha valore NIL se il suo argomento è T. Se il suo argomento è qualcosa di diverso da T o NIL, ha valore NIL. Basta la semplice osservazione per notare che l'operatore NOT inverte il suo argomento per formare il suo valore. Va notato anche che NULL e NOT sono esattamente identici, ma il Lisp li mantiene entrambi (fig. 3.6).

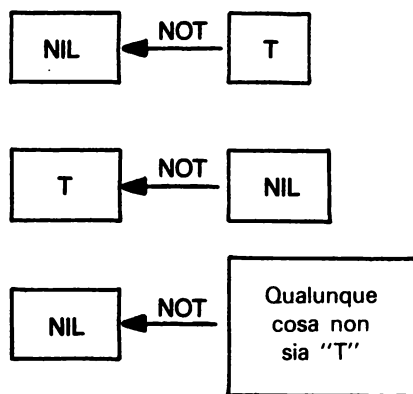



Fig. 3.6.

Che cosa si può dire degli operatori AND *e* OR*?*
Gli operatori AND e OR possono avere un numero indefinito di argomenti. AND (che corrisponde all'incirca alla congiunzione "e") è sempre NIL, "a meno che" tutti i suoi argomenti siano diversi da NIL. Qui, come prima, NIL significa "falso" e diverso da NIL significa "vero". L'operatore OR è sempre vero a meno che tutti i suoi argomenti siano NIL. Se tutti gli argomenti di AND sono veri, allora il valore fornito da

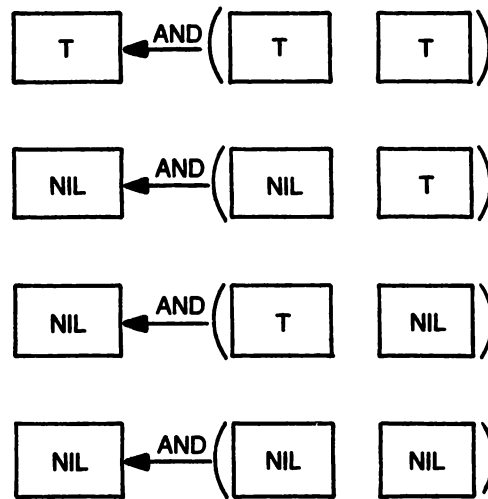



Fig. 3.7.

AND è T. Analogamente, il valore dell'operatore OR è NIL se tutti i suoi argomenti sono NIL, T altrimenti.

Qualche esempio dell'operatore AND *(fig. 3.7)?*

```
(AND T T) è T
(AND NIL T) è NIL
(AND T NIL) è NIL
(AND NIL NIL) è NIL
(AND T T T T) è T
(AND NIL NIL NIL) è NIL
(AND NIL NIL T T T T) è NIL
```

Qualche esempio dell'operatore OR *(fig. 3.8)?*

```
(OR T NIL) è T
(OR T T) è T
(OR NIL T) è T
(OR NIL NIL) è NIL
(OR NIL NIL T T NIL) è T
(OR NIL NIL NIL NIL NIL) è NIL
(OR NIL T NIL NIL NIL NIL) è T
```

Si può usare l'operatore OR *per creare una funzione?*
Sì. Un ottimo esempio si ha usando l'OR per migliorare la funzione MEMBER. La funzione MEMBER, ora, può determinare solo se il suo

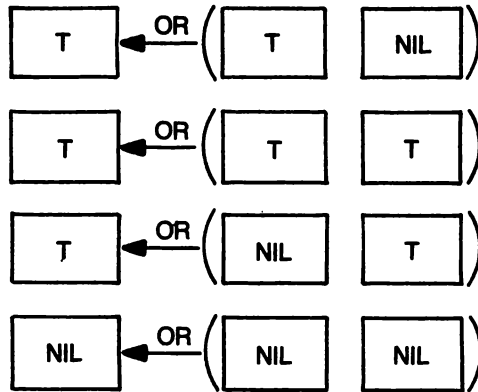



Fig. 3.8.

primo argomento è membro della lista data. Usando OR si può estendere il campo d'azione della funzione MEMBER in modo da poter trattare liste che contengano sottoliste.

Come si scriverebbe questa versione migliorata della funzione MEMBER*?*
La si può scrivere così:

```
(DEFINE ((
(MEMBER (LAMBDA (A B)
(COND((NULL B)
NIL)(T OR(COND((ATOM(CAR B))
(EQUAL A(CAR B)))
(T(MEMBER A(CAR B))))
(MEMBER A(CDR B)))) )))))
```

AND, NOT *e* OR *possono essere visti come dei predicati?*
Sì, perché i loro valori sono sempre T o NIL. Tuttavia sono diversi dagli altri predicati perché anche i loro argomenti, come i loro valori, sono normalmente o T o NIL.

NOTAZIONE CON IL PUNTO

Che cos'è la notazione con il punto?
La lista (X. Y) in Lisp è una lista di due elementi, ma non va confusa con la lista (X Y). Quando sono scritti nella forma (X. Y) i due elementi X e Y sono dei puntatori.

Si può chiarire meglio questo concetto di puntatore?

Per capire le coppie puntate, cioé quelle della forma (X. Y), bisogna sapere come vengono trattate le liste nella memoria del computer. La memoria è il meccanismo che immagazzina attivamente le liste durante l'esecuzione del programma. In Lisp, una lista è formata da una sequenza di coppie. Il primo elemento di ogni coppia è un puntatore a un atomo o a una sottolista (una lista elemento di una lista). Il secondo elemento di ciascuna coppia è un puntatore alla successiva coppia di elementi. La *notazione con il punto* consente di considerare le coppie separatamente (fig. 3.9).

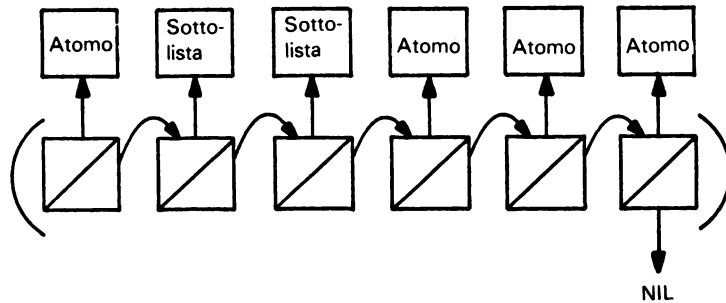



Fig. 3.9. NIL è un puntatore speciale che punta alla fine della lista.

Che cos'è una coppia puntata?
(X. Y) è una singola coppia. Il suo primo elemento, X, è un puntatore a un atomo o a una sottolista chiamata X. Il suo secondo elemento è un puntatore alla successiva coppia Y.

Come viene formata la lista (X Y) *con l'uso delle informazioni date?*
La lista (X Y) verrebbe formata in un modo molto diverso. Innanzitutto, è necessario che siano chiamate in gioco due coppie. La prima coppia sarebbe il puntatore a X, la seconda sarebbe un puntatore a Y. Non bisogna dimenticare poi che il primo puntatore deve puntare anche alla seconda coppia. Ovviamente è il secondo elemento della prima coppia che punta alla seconda coppia. Anche il secondo elemento della seconda coppia deve puntare alla coppia successiva, se si prende alla lettera la definizione data prima. In molti casi, il secondo elemento conterrà zero, in particolare in molte situazioni di elaborazione di liste.

Perché il secondo elemento dovrebbe contenere zero?
Per indicare la fine di una lista. In Lisp, il secondo elemento non conterrà mai in effetti zero, ma invece NIL, che fungerebbe da puntatore speciale che punterebbe alla fine di una lista. Questo puntatore speciale scritto in notazione con il punto sarebbe:

(Y.NIL).

Come verrebbe mostrata una lista?
Nella notazione con il punto, si può scrivere:

(X.(Y.NIL)).

Come si scriverebbe una lista che contenesse sottoliste, nella notazione con il punto?
Si scriverebbe così:

(Q.((W.(E.NIL)).NIL))

assumendo che la lista fosse:

(Q(W E)).

Come si interpreta questa scrittura relativa alle sottoliste, nella notazione con il punto?
La prima coppia sarebbe costituita dal puntatore a Q e da un puntatore alla seconda coppia. La seconda coppia sarebbe costituita da un puntatore a (W E) e da un puntatore a NIL.

Che cosa si intende per "puntatore a (W E)"*?*
Si intende un puntatore alla prima coppia nella rappresentazione di (W E). Questa coppia, ovviamente, è formata da un puntatore a W e da un puntatore alla seconda coppia.

Tutto questo è molto importante per il Lisp?
La convenzione che un puntatore alla prima coppia di una serie di coppie che rappresentano una lista può essere preso come puntatore alla lista stessa porta a una spiegazione e una comprensione corrette delle funzioni fondamentali CAR, CONS e CDR.

LA LISTA GENERALE

Che cos'è una lista generale in Lisp?
Una qualunque lista nella forma:

(A B C D E F G H I J K)

dove ciascuno dei simboli A B C D E F G H I J K può stare per un atomo o una sottolista. La prima coppia nella rappresentazione di questa lista è formata da un puntatore ad A e da un puntatore alla seconda coppia. Le altre coppie, a partire dalla seconda, sono esattamente nello stes-

so formato che avrebbero se la lista fosse:

(B C D E F G H I J K)

anziché

(A B C D E F G H I J K).

Un puntatore alla seconda coppia nella lista originale pertanto può essere considerato un puntatore alla lista:

(B C D E F G H I J K)

o alla rappresentazione della lista.

Qual è la definizione generale di lista?
Una lista è rappresentata in memoria da una coppia formata da un puntatore a (CAR A) e un puntatore a (CDR A), assumendo che il nome della lista sia A. Poiché la definizione è ricorsiva, non si deve interpretare questa affermazione nel senso che una lista sia rappresentata da una e una sola coppia. (CAR A) e (CDR A) saranno rappresentati a loro volta da coppie. Questa definizione è il motivo effettivo per cui CAR e CDR sono così importanti nel Lisp (fig. 3.10).

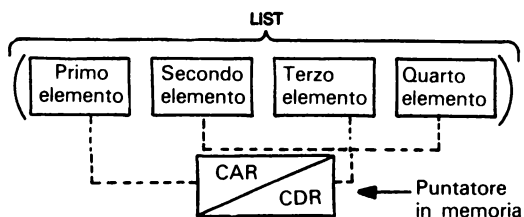



Fig. 3.10. Ogni lista può essere pensata come la coppia che punta al CAR e al CDR della lista stessa.

Come si possono rappresentare le funzioni CAR *e* CDR *nella notazione con il punto?*

Se Z è (X. Y), allora (CAR Z) è X e (CDR Z) è B

e inoltre

se Z è (X. Y), allora (CONS X Y) è C.

Quando si può usare la notazione con il punto in un programma in Lisp?
La notazione con il punto può essere usata ovunque in una funzione o in un programma in Lisp, per la creazione di liste temporanee e per vari altri scopi. Usa una minore quantità di memoria che non la notazione a liste, ma bisogna ricordare che ogni volta possono essere "puntate" solo due grandezze. Queste espressioni non hanno assolutamente significato in Lisp:

```
(S.D.P)
(X.Y.Z)
```

ma, come abbiamo visto prima, si deve scrivere:

```
(S.(D.P.))
(X.(Y.Z)).
```

Ogni lista esistente nel Lisp che possa essere espressa come una S-espressione può essere trasformata nella notazione con il punto.
Attenzione! Non tutte le espressioni nella notazione con il punto possono essere espresse nella notazione a liste (la notazione standard usata nelle S-espressioni).

RICORSIONE A DUE LISTE

Che differenza c'è fra la ricorsione a due liste e la ricorsione a lista singola?
La costruzione di funzioni per la ricorsione a due liste è identica alla costruzione di funzioni con l'uso di una sola lista.

Che cosa fa la funzione del sistema Lisp APPEND*?*
Compone due liste, come nei casi che seguono:

se A è (3 4 5) e B è (4)
se C è (7 6) e D è (7 6 5)
se E è (4 5 6) e F è (3)

allora

```
(APPEND A B) è (3 4 5 4)
(APPEND C D) è (7 6 7 6 5)
(APPEND E F) è (4 5 6 3)
```

Come si definisce APPEND *in Lisp?*

```
DEFINE((
(APPEND(LAMBDA (X Y)
(COND((NULL
X)Y)(T(CONS(CAR X)
(APPEND (CDR X)
Y))) )))))
```

Si può ridefinire EQUAL *usando l'idea della ricorsione a due liste?*

```
DEFINE((
(EQUAL(LAMBDA (X Y)
(COND((NULL X)
(NULL Y))((ATOM X)(AND
(ATOM Y)(EQ X Y)))
((ATOM Y)NIL)(T
(AND(EQUAL(CAR X)
(CAR Y))
(EQUAL(CDR X)(CDR Y))))
```

Questo è un metodo più semplice e più veloce che funziona solo sugli atomi rispetto a EQUAL.

FUNZIONI DI TIPO

Che cos'è una funzione di tipo?
Abbiamo già usato una funzione di tipo, la funzione ATOM. Fondamentalmente, una funzione di tipo determina di quale tipo sia una variabile, un numero o un altro oggetto.

Come si stabilisce se una variabile è un numero o no?
Si usa il predicato NUMBERP.

Si può avere qualche esempio dell'uso di NUMBERP*?*

Se A è 5
se C è T (il simbolo)
se H è (T)
se J è (4 5 6)
se K è 2345

allora

(NUMBERP A) è T
(NUMBERP C) è NIL

(NUMBERP H) è NIL
(NUMBERP J) è NIL
(NUMBERP K) è T.

NUMBERP *si applica a numeri e* ATOM *ad atomi. Vi sono altre funzioni di "tipo"?*
Sì, vi sono due altre funzioni di tipo, e ambedue hanno a che fare con i numeri. Sono FIXP e FLOATP.

Che cosa fa FIXP*?*
Determina se un numero è in virgola fissa (intero) o no.

Qualche esempio?

Se A è 34.9
se B è 45
se C è 56.98
se D è 2345.9
se E è 34

allora

(FIXP A) è NIL
(FIXP B) è T
(FIXP C) è NIL
(FIXP D) è NIL
(FIXP E) è T.

Che cosa fa FLOATP*?*
FLOATP controlla se un numero è in virgola mobile o meno. Controlla se c'è qualcosa alla destra del punto decimale o meno, "se si tratta di un intero o no". Sostanzialmente, è l'opposto di FIXP.

Qualche esempio di FLOATP*?*

Se A è 89
se B è 5.87
se C è 76
se D è 56.03
se E è 2345

allora

(FLOATP A) è NIL

(FLOATP B) è T
(FLOATP C) è NIL
(FLOATP D) è T
(FLOATP E) è NIL.

C'è qualche funzione importante che possa essere costruita da FLOATP *e* FIXP*?*
Sì, si chiama COUNTFIXFLOAT. COUNTFIXFLOAT dà come valore una lista di due interi. Il primo intero della lista è il numero dei numeri in virgola fissa nella lista che è argomento di COUNTFIXFLOAT; il secondo intero rappresenta il numero dei numeri in virgola mobile presenti nella lista argomento della funzione.

Si può avere qualche esempio del modo in cui agisce questa funzione?

Se A è (45 3.8)((34.9 56)(56.9))8)
se B è (4.9 67 34.98 6.9 89 7)
se C è (34 (56.9 56.8)4)
se D è (4(6(56.8(65.9(56)))))

allora

(COUNTFIXFLOAT A) è (3 3)
(COUNTFIXFLOAT B) è (3 3)
(COUNTFIXFLOAT C) è (2 2)
(COUNTFIXFLOAT D) è (3 2)

Come si scriverebbe l'espressione definitoria per COUNTFIXFLOAT*?*

```
DEFINE((
(COUNTFIXFLOAT(LAMBDA (X)
(COND((NULL X)
(QUOTE (OO)))
((ATOM X)(COND((NOT
(NUMBERP X))(QUOTE (O O)))
((FIXP X)(QUOTE (O 1)))(T
(QUOTE (O O)))))
(T(SUMM(COUNTFIXFLOAT(
CAR X)) (COUNTFIXFLOAT(
CDR X)))) )))
```

Che cos'è la funzione chiamata SUMM*?*
È una subroutine usata da COUNTFIXFLOAT. Questa funzione somma le due liste di numeri (interi e non) elemento per elemento. Inoltre deve essere inclusa con la definizione di COUNTFIXFLOAT. Per concludere la definizione di COUNTFIXFLOAT, dunque, ecco la parte che si riferisce

alla funzione SUMM:

```
(SUMM(LAMBDA(X Y)(CONS
(PLUS (CAR X)
(CAR Y))(LIST(PLUS(CADR X)
(CADR Y)))) )))).
```

Si noti che questa espressione è parte integrante della definizione di COUNTFIXFLOAT e non deve essere separata, quando viene usata nella pratica effettiva. Nell'usare questa funzione COUNTFIXFLOAT, la subroutine SUMM va scritta così com'è direttamente dopo l'ultima riga dell'espressione precedente.

ANCORA UN RIPASSO

Nell'esempio che segue, quali variabili sono vincolate e quali sono libere?

(LAMBDA(X Y Z)(TIMES(PLUS X Z)(PLUS Y W)(MINUS U)))

X Y e Z sono vincolate, mentre W e U sono libere.

Qual è il valore di (CAR(QUOTE(A B C)))?
A.

Qual è il valore di (PLUS 3(CAR(QUOTE(5 6))))?
8.

Qual è il valore di (CDR(LIST(QUOTE(4 6))3))?
(3 9 7 4).

Qual è il valore di (CONS(QUOTE TOP)(LIST (QUOTE I)))?
(TOP I).

Se si applica la funzione CDDR *a* (4 5 6 7), *che valore dà?*
(6 7).

Se si applica la funzione CAADR *a* (6((7)2 4)), *che valore dà?*
7.

Traducete nella notazione con il punto ((X)Y).
((X.NIL).(Y.NIL)).

Traducete nella notazione con il punto (X(Y)Z(A B)).
(X.((Y.NIL).(Z.((A.(B.NIL)).NIL)))).

Traducete nella notazione con il punto (((Q)R S)T U).
(((Q.NIL).(R.(S.NIL))).(T.(U.NIL))).

Traducete nella notazione a lista (S.(T.(U.NIL))).
(S T U).

Traducete nella notazione a lista (NIL. (Y.NIL)).
(NIL Y).

Traducete nella notazione a lista (Q.(W.((E.(R.((T.NIL).NIL))).NIL))).
(QW(ER(T))).

Qual è il valore di (NUMBERP(DIFFERENCE 10(QUOTIENT 20 4)))*?*
T.

Qual è il valore di (NUMBERP(CAR(QUOTE(PLUS 3 4))))*?*
NIL.

Qual è il valore di (NUMBERP –9)*?*
T.

CAPITOLO QUATTRO

La programmazione in Lisp

Come si scrive un programma in Lisp?
Lo si scrive in modo che sostituisca una funzione o la descrizione di una funzione.

E come si fa?
Usando la speciale funzione PROG (in Lisp tutto viene fatto mediante funzioni) che può avere un numero indefinito di argomenti.

Dove si trova di solito la funzione PROG*?*
Una espressione PROG in genere costituisce il secondo argomento di una espressione LAMBDA, ma può essere usata anche altrove.

Quali sono gli argomenti di una espressione PROG*?*
Sono dichiarazione (*declaration*), enunciati (*statements*), etichette (*labels*), trasferimenti (*transfers*), condizionali (*conditionals*) e valori in uscita (*returning values*) (fig. 4.1).

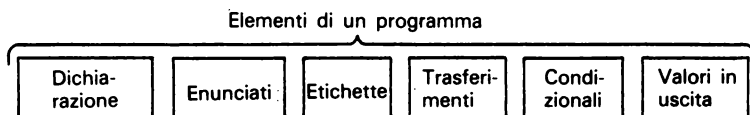



Fig. 4.1.

Che cosa sono le dichiarazioni?
Sono il primo argomento di PROG e costituiscono un elenco delle variabili che si presentano in quel PROG. Queste variabili sono chiamate variabili di programma e debbono essere definite, ma non come variabili intere o in virgola mobile, ecc. Le variabili cambiano spesso tipo durante l'esecuzione del programma. Come al solito, qualunque variabile può rappresentare una lista.

Che cosa sono gli enunciati?
Gli enunciati sono espressioni come le assegnazioni (SETQ, SET), le

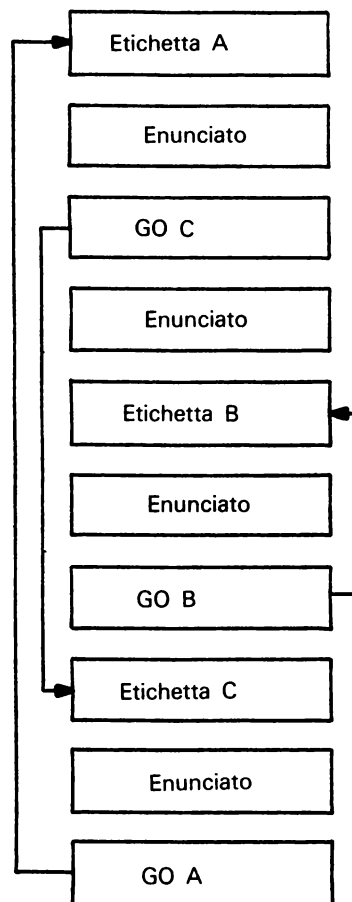



Fig. 4.2. Bisogna ricordare che una funzione GO può essere l'azione di un predicato. Il trasferimento può essere ottenuto con i risultati raggiunti da una condizione come nel caso di (GO B) e (GO C), mentre (GO A) può essere un trasferimento incondizionato all'etichetta A. Si può ipotizzare che nel programma, in qualche punto prima di (GO A) esista un'uscita, altrimenti si ha un ciclo infinito.

chiamate di subroutine. Queste sono semplicemente espressioni che usano il nome della subroutine come funzioni nelle espressioni.

Che cosa sono le etichette?
Le etichette sono sempre atomi singoli e si distinguono dalle funzioni proprio per il fatto che sono atomi.

Che cos'è un trasferimento?
Un trasferimento è come un enunciato GO TO. In Lisp, è la funzione GO e ha un solo argomento, un'etichetta (fig. 4.2). Per esempio:

```
(GO RESULTS)
(GO START)
(GO DIGIT)
```

Che cosa sono i condizionali?
Ricordate la funzione COND, analoga agli enunciati IF di altri linguaggi. COND è diversa nelle funzioni e nei programmi. Nelle funzioni ha come secondo elemento di ciascuna coppia una quantità, mentre in un programma il secondo elemento di ciascuna coppia è un enunciato (qualsiasi enunciato che sia lecito in Lisp). In un programma, poi, l'ultima coppia di elementi non deve essere una T. Questo significa che se tutte le coppie di elementi (argomenti) sono NIL, allora si esegue l'enunciato successivo.

Che cos'è un valore in uscita?
Per avere un valore in uscita, si usa la funzione RETURN, che ha solo un argomento, il valore che deve essere "ritornato". Per esempio:

```
(RETURN K).
```

Si può vedere un programmino che usi tutti questi elementi di cui abbiamo appena parlato?
Sì, possiamo scrivere questo programma che definisce i fattoriali:

```
DEFINE(( (FATT(LAMBDA (N) (PROG (I J)
(SETQ I N) (SETQ J 1)
K(COND((ZEROP I)(GO L)))
(SETQ J(TIMES J I))
(SETQ I (DIFFERENCE I 1)
(GO K)
L(RETURN J) ))))
```

Vanno notate le cinque parentesi chiuse dopo l'ultimo enunciato. In generale si usano tre parentesi se debbono entrare altre definizioni nella stessa espressione DEFINE e cinque parentesi negli altri casi.

ANCORA SULLA PROGRAMMAZIONE

Come si costruiscono le liste?
Quando si scrive un programma in Lisp che produce come valore una lista, è importante che la lista sia prodotta nell'ordine giusto e nella giusta direzione. È più efficace, nella costruzione di liste, cominciare con l'ultimo elemento e aggiungere davanti ad esso gli altri elementi, per finire con il primo della lista.
Il motivo è che il Lisp tiene sempre conto del punto in cui si trova l'inizio di ogni lista, ma non del punto in cui si trova la fine di ogni lista. Se si devono inserire elementi alla fine di una lista, si deve percorrere la lista dal primo elemento in avanti, cercando l'ultimo. Se non si sa dove si trovi l'ultimo elemento, non si può posizionare qualcosa dopo di esso.
La funzione usata comunemente per aggiungere elementi a una lista è CONS. È meglio usare CONS, anziché APPEND, nella scrittura dei programmi. È vero che APPEND può posizionare elementi sia all'inizio sia alla fine di una lista, ma CONS effettua l'operazione più rapidamente e li colloca all'inizio della lista. In effetti, CONS è molto più rapida di APPEND. Per usare APPEND:

se S è (4) e Q è (2 3)

allora

(APPEND Q S) è (2 3 4)

mentre

(APPEND S Q) è (4 2 3)

e usando CONS scriviamo:

(CONS S Q)

che ci dà (4 2 3).
Tuttavia, bisogna ricordare che CONS non può essere usata direttamente al posto di APPEND.

E perché no?
Perché il primo argomento di APPEND è sempre una lista, mentre il primo argomento di CONS di solito non è una lista.

Si può scrivere un programma che inverte una lista?
Sì, usando la funzione CONS:

```
DEFINE ((
(INVERSA(LAMBDA (A)
(PROG (X Y)
(SETQ X A)(SETQ Y NIL)
K(COND((NULL X)(RETURN Y)))
SETQ Y(CONS
(CAR XY))
(SETQ X (CDR X))
(GO K) )))))
```

Non è un po' curiosa questa situazione?
Beh, sì, il modo più efficiente per creare una lista è procedere dalla fine all'inizio, ma per usarla il modo più efficiente è procedere dall'inizio verso la fine (fig. 4.3).

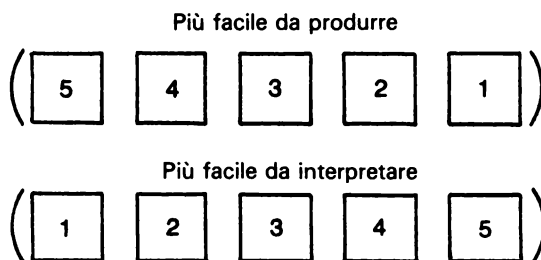



Fig. 4.3.

Qual è il nome delle funzioni del Lisp che possono costruire liste da liste?
Sono le funzioni MAP. MAPCAR è una funzione a due argomenti, il primo dei quali denota una lista mentre il secondo è il nome di un'altra funzione. MAPCAR applica la funzione citata agli elementi della lista e produce come valore una lista dei risultati. Specificamente, MAPCAR applica la funzione data come secondo argomento a:

```
(CAR L)
(CADR L)
(CADDR L)
(CADDDR L)
(CADDDDR L)
(CADDDDDR L)
```

e via dicendo, dove L è la lista che compare come primo argomento.

Vi sono altre funzioni MAP*?*
Sì, per esempio MAPLIST. Anche MAPLIST ha due argomenti, il primo dei quali deve essere una lista e il secondo una funzione che deve essere

applicata agli elementi della lista. Tuttavia c'è una differenza: MAPLIST applica prima la funzione alla lista stessa, poi a

```
(CDR L)
(CDDR L)
(CDDDR L)
(CDDDDR L)
(CDDDDDR L)
```

e via dicendo.

Come si possono rappresentare, in Lisp, le matrici?
Usando una lista di liste. Se si vuole la matrice:

```
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 1
3 4 5 6 7 8 9 1 2
4 5 6 7 8 9 1 2 3
5 6 7 8 9 1 2 3 4
6 7 8 9 1 2 3 4 5
7 8 9 1 2 3 4 5 6
8 9 1 2 3 4 5 6 7
9 1 2 3 4 5 6 7 8
```

che è una matrice 9 × 9, si può scrivere:

```
((1 2 3 4 5 6 7 8 9)(2 3 4 5 6 7 8 9 1)(3 4 5 6 7 8 9 1 2)
 (4 5 6 7 8 9 1 2 3)(5 6 7 8 9 1 2 3 4)(6 7 8 9 1 2 3 4 5)
 (7 8 9 1 2 3 4 5 6)(8 9 1 2 3 4 5 6 7)(9 1 2 3 4 5 6 7 8))
```

Si può scrivere un programma che realizzi una lista del genere come nell'esempio precedente?
Sì, usando la funzione GREATERP:

```
DEFINE((
(MAT(LAMBDA (N) (PROG
(I J K L Z)
SETQ J N)(SETQ K NIL)
B(COND((EQUAL I J)
(SETQ Z 1))
(T(SETQ Z 0)))
SETQ K(CONS Z K))
(COND((GREATERP J 0)
(GO B)))
(SETQ L (CONS K L))
((SETQ I (DIFFERENCE I 1))
```

```
(COND((GREATERP I 0)
(GO A)))
(RETURN L) )))))
```

Qual è la differenza fra l'effetto di una funzione e il suo valore?
L'effetto è ciò che la funzione causa alle variabili: per esempio, può modificarne il valore. Il valore della funzione invece è il valore che la funzione dà senza modificare i valori delle variabili che ha utilizzato. In Lisp, molte fra le funzioni tipiche possono essere utilizzate sia per il loro effetto sia per il loro valore. Possono essere usate, cioé, per modificare il valore di qualche variabile oppure possono essere usate per produrre qualche valore. Per esempio, le espressioni che seguono usano funzioni tipiche solo per produrre un valore:

```
(CAR L)
(CDR L)
(CONS E R)
(DIFFERENCE K L)
(TIMES F G)
(PLUS J K)
(QUOTIENT I P)
```

In ciascuno di questi casi le variabili usate dalle funzioni non vengono modificate, ma si notino le funzioni (espressioni) seguenti:

```
(SET A B)
(SETQ R T).
```

Queste funzioni modificano il valore delle variabili che usano. In altre parole, hanno un effetto sulle variabili.

Si possono usare i predicati nei programmi?
Sì, quando un predicato viene usato in un programma ritorna semplicemente un valore T o NIL. I programmi che danno in uscita T e NIL con tutta probabilità daranno T in certe occasioni e NIL in altre. Ovviamente, in Lisp, come in altri linguaggi di programmazione, si possono avere più funzioni di uscita.

Che cos'è un programma ricorsivo in Lisp?
Un programma ricorsivo in Lisp è come qualunque altro programma ricorsivo in qualunque altro linguaggio che consenta la ricorsività. Un programma ricorsivo è o un programma che chiama se stesso, o un programma all'interno di un gruppo di programmi che si chiamano l'uno dopo l'altro in modo ciclico.
Rispetto a quel che accade con altri linguaggi, è facile che i programmi

in Lisp siano ricorsivi, principalmente perché il Lisp agisce su liste. Dovrebbe essere abbastanza comprensibile che i programmi in Lisp che agiscono su liste "non sono così ricorsivi" come le funzioni stesse che agiscono su liste, perché un programma in Lisp di solito userà un ciclo, laddove la corrispondente funzione userebbe, allo stesso fine, una ricorsione. Tuttavia, la manipolazione delle sottoliste da parte di un programma (che assume l'esistenza di sottoliste di una lista) di solito è esplicitamente ricorsiva. Un corrispondente programma non ricorsivo richiederebbe l'uso di una lista del tipo *push-down*, cioé "a catasta" e pertanto sarebbe più contorto.

Si può riscrivere la funzione ADL *(vedi pag. 85) come programma?*
Sì, e in questo modo si può anche ridurre un livello di ricorsività:

```
(DEFINE((
(ADL(LAMBDA (L)(PROG (M N X)
(SETQ M L) (SETQ N 0)
A(COND((NULL M)
(RETURN N)))
(SETQ X (CAR M))
(SETQ M (CDR M))
(COND((ATOM X)
(SETQ N (PLUS N X)))
(T(SETQ N(PLUS N(ADL X)))))
(GO A) ))))).
```

Questo è un tipico programma Lisp.

Si può scrivere un programma all'interno di un altro programma?
Sì, però ogni volta che si usa PROG il sistema usa una certa quantità di tempo, che è inevitabile. Anche se non vi sono nuove variabili, il sistema deve comunque controllare. Inoltre, questo non è un metodo di programmazione particolarmente efficiente. In Lisp esiste un'altra funzione chiamata PROGN, simile a un enunciato composto in Algol. Sempre per conservare questo paragone con l'Algol, la funzione PROG del Lisp è simile a un blocco in Algol.

Che cos'è PROGN?
PROGN ha un numero di argomenti indefinito. Ciascun argomento è un enunciato che rappresenta un programma Lisp. L'unico obiettivo di PROGN è riunire questi enunciati in un enunciato unico. Vi è una restrizione importante: non è ammesso l'uso di "etichette". Come ogni altra funzione nel Lisp, PROGN ha un valore, che è *sempre* il valore del suo ultimo argomento. Quindi è facile vedere che PROGN può essere usata non solo per il suo effetto, ma anche per il suo valore. Questa funzione PROGN consente di formare espressioni del tipo WHERE, presenti in alcuni linguaggi, ma non in Fortran né in Algol.

Che cos'è la funzione PROG2*?*
PROGN è un'estensione della funzione PROG2, che ha solo due argomenti. Come per PROGN, il valore di PROG2 è quello del suo ultimo argomento. Alcuni sistemi Lisp hanno la funzione PROG2 ma non PROGN.

Si può avere un esempio dell'uso di PROGN*?*
Per chiarire come opera, scriviamo un programma più complesso, anche se non più lungo. Lo chiameremo MAX. MAX accetterà due liste, una delle quali numerica, mentre l'altra è una lista di atomi (la lista numerica in effetti è una lista di atomi numerici). MAX stabilirà quale sia il maggiore fra i numeri nella prima lista, poi sceglierà nella seconda lista di atomi l'atomo che occupa la medesima posizione di quel numero. Per esempio, se la lista numerica è:

(4 6 3 8 9)

il valore massimo è 9 e si trova nella quinta posizione.
Abbiamo bisogno anche di una lista di atomi, per esempio:

(L I B R O)

Il programma associerà al 9 la lettera O perché anche questa occupa la quinta (e ultima) posizione nella rispettiva lista.
Ecco dunque il programma:

```
DEFINE((
(MAX(LAMBDA (J K)(PROG
(T U R S W X)
(SETQ T (CAR J))
(SETQ U (CAR K))
(SETQ R (CDR J))
A(COND((NULL R)
RETURN U)))
(SETQ W (CAR R))
(SETQ X (CAR S))
(SETQ R (CDR R))
(SETQ S (CDR S))
(COND((GREATERP W T)
(PROGN(SETQ T W)(SETQ U X))))
(GO A) )))))
```

Ovviamente, in questo programma PROGN potrebbe essere sostituita da PROG2.

Che cosa fa PROGN*?*
La funzione PROGN valuta i suoi argomenti.

Che cosa vuol dire valutare una funzione?
Quando si dice che si valuta TIMES (9 7) si vuol dire in effetti che se ne cerca il valore. In Lisp, quando si parla di valutare una funzione si vuol dire "eseguire" quella funzione. Questa distinzione è necessaria perché le funzioni possono essere usate anche per il loro effetto. Se si deve valutare l'espressione

(SETQ A B)

non si chiede semplicemente il suo valore, che ovviamente è il valore di B, ma si vuole il suo effetto. L'espressione

(SETQ A B)

ha l'effetto di attribuire ad A il valore di B.
Se invece si parla di una funzione aritmetica, quando si parla di valutarla si intende probabilmente dire che se ne vuol trovare il valore.

Che cos'è la regola di valutazione in Lisp?
Per valutare una funzione F nel modo normale, i suoi argomenti, atomi e funzioni, vengono valutati nell'ordine, da sinistra verso destra. La definizione di F viene applicata a questa lista di valori, e dà il valore di F.

Perché bisogna stare molto attenti all'uso di AND*?*
Perché la funzione AND non è commutativa.

Che cosa ha a che vedere la valutazione con la funzione DEFINE*?*
La regola di valutazione vale per tutte le funzioni e i programmi definiti mediante la funzione DEFINE. Ammette però qualche eccezione importante.

Quali sono queste eccezioni?
AND e OR vengono valutate in modo diverso. Quando si valuta

(AND A B)

si valuta A e poi si controlla se è NIL. In caso positivo, il valore di (AND A B) è NIL e non c'è bisogno di valutare B. Solo se il valore di A è T si deve valutare B.

Perché non si può usare AND*, al posto di* PROGN *o* PROG2*, in un programma?*
Se si scrive:

(AND(SETQ A B)(SETQ C D))

invece di

(PROG2(SETQ A B)(SETQ C D))

per lo più non ci sarebbero problemi, ma se B dovesse avere valore NIL, il programma non arriverebbe mai a

(SETQ C D).

E che cosa mi dici di OR*?*
La valutazione di OR procede all'inverso di AND. Se si scrive:

(OR A B)

si valuta prima A e si controlla se il suo valore è NIL. Se è NIL si valuta B, altrimenti se A è T, l'espressione (OR A B) è vera e non c'è bisogno di valutare B.

Qual è la regola per AND *e* OR*?*
L'uso di AND e OR, con più di un argomento, procede da sinistra verso destra fino a che non si trova un argomento NIL (per una espressione AND) o un argomento T (per una espressione OR) (fig. 4.4).

Fig. 4.4.

E che cosa si può dire di altre valutazioni?
Se si ha una variabile il cui valore è una lista, con una S-espressione indicante che può essere trattata come un uso di una funzione (il primo elemento della lista è un nome di funzione e gli altri elementi sono argomenti) e si vuole valutare quella funzione, cioé trovare il valore della funzione applicata a quegli argomenti, si può usare la funzione EVAL.

Qualche esempio di EVAL*?*
Se A è (PLUS 3 4)

se B è (DIFFERENCE 6 3)
se C è (LIST 5 6 7 8)
se D è (QUOTE TOPI)

allora

(EVAL A) è 7
(EVAL B) è 3
(EVAL C) è (5 6 7 8)
(EVAL D) è TOPI.

C'è una differenza fra (EVAL (PLUS 3 4)) *e* (EVAL A), *dove* A *è* (PLUS 3 4)*?*
Sì, c'è una differenza. Perché

(EVAL(PLUS 3 4))

ha in effetti il valore 7, ma non per motivi ovvi. Se si scrive

(DIFFERENCE 10(PLUS 3 4))

la risposta, 3, è identica a quella che si ottiene per

(DIFFERENCE 10 7)

e in un certo senso

(EVAL(PLUS 3 4))

è come

(EVAL 7).

Quando si valuta un'espressione come

(EVAL 7)

si ottiene quella che sembra la risposta corretta perché il valore di 7 è ovviamente 7. (Il valore di un qualunque intero è l'intero stesso.)
Se si scrive:

(EVAL(LIST(QUOTE MINUS)6))

il valore che si ottiene è –6 (non (MINUS 6)) perché il valore di

(LIST(QUOTE MINUS)6)

è (MINUS 6) e il valore di questa espressione è −6. Per evitare qualunque confusione usando la funzione EVAL si può scrivere:

(EVAL(QUOTE(PLUS 3 4)))

quando si vuol fare riferimento a

(EVAL A)

dove A sta per (PLUS 3 4).

Esiste qualche funzione analoga a EVAL, *in altri linguaggi?*
Nella maggior parte degli altri linguaggi di programmazione no. Fortran, Basic, PL/1 e Algol non hanno nulla del genere.

Perché EVAL *è importante?*
La funzione EVAL rende il Lisp molto più potente della maggior parte degli altri linguaggi algebrici. Con EVAL si possono eseguire enunciati costruiti dal programma Lisp. Queste costruzioni possono essere diverse, ogni volta che si fa girare il programma. Nella maggior parte dei linguaggi algebrici, la forma degli enunciati deve essere costruita all'inizio del programma e non può essere mutata a metà. In particolare, non può essere mutata durante l'esecuzione di un programma, mentre in Lisp è possibile modificare gli enunciati nel corso dell'esecuzione di un programma. Questa caratteristica rende il Lisp molto interessante, quando si studiano l'intelligenza artificiale o l'euristica (programmi ad autoapprendimento), perché i programmi possono modificarsi da soli durante l'esecuzione.

Qualche linguaggio ha una funzione analoga a EVAL*?*
Sì, lo Snobol 4 ha la funzione CODE, che è come EVAL.

Che cos'è EVALQUOTE *in Lisp?*
EVALQUOTE è simile a EVAL, solo che ha due argomenti invece di uno solo. Il primo argomento è il nome di una funzione, il secondo è una lista degli argomenti della funzione. Abbiamo qui un'altra eccezione alla regola generale di valutazione.

Perché?
Gli argomenti di EVALQUOTE non vengono valutati, ma vengono assunti esattamente nella forma in cui sono "citati". La funzione EVALQUOTE "cita" i suoi argomenti, come la funzione SETQ, che cita solamente i suoi argomenti, anziché valutarli.

Questo aiuta a spiegare l'uso di funzioni a livello 0?
Sì, il motivo alla base dello speciale trattamento delle parentesi al livello 0 è una conseguenza diretta di quanto abbiamo appena detto. La funzione EVALQUOTE è costruita in parte per le particolari esigenze del Lisp, rispetto al sistema Lisp che legge le funzioni e i loro argomenti da schede o da un terminale. Le due S-espressioni che vengono lette contemporaneamente da questa parte del sistema (il "supervisore" Lisp) sono esattamente i due argomenti di EVALQUOTE. In Lisp, il supervisore è un programma estremamente semplice, mentre altri linguaggi hanno bisogno di supervisori molto complessi. Tutto ciò che il supervisore deve fare, in Lisp, è chiamare continuamente la funzione EVALQUOTE. Ogni volta EVALQUOTE legge una coppia di argomenti dal dispositivo di input. Questo supervisore è detto "supervisore EVALQUOTE" ed è disponibile su quasi tutti i sistemi Lisp. Qui si vede bene come il Lisp sia un vero linguaggio funzionale: lo stesso supervisore è stato costruito come una funzione Lisp.

Che cos'ultro può fare la funzione EVAL*?*
La funzione EVAL può valutare non solo enunciati singoli, ma addirittura programmi interi. Di solito un programma è piuttosto lungo e complesso. Un unico enunciato che costruisca un programma a partire da una struttura a lista molto grande è difficile da analizzare. Per semplificare le cose, si possono usare variabili per rappresentare strutture a liste. Queste variabili possono essere combinate in modo che le strutture a lista che rappresentano diventino sempre più grandi fino a che non si è completato l'"enunciato" del programma a partire da queste variabili.

Che cos'è la funzione FOR *in Lisp?*
La funzione FOR è analoga all'enunciato DO in Fortran e all'enunciato FOR in Algol. Si scrive:

(FOR A B C D DESCRIZIONE)

Il valore della funzione FOR è una lista di enunciati che apparirebbero in un programma Lisp e che corrisponde a ripetere la lista di enunciati che è il valore di DESCRIZIONE per A a partire dal valore B, per crescere fino a un valore massimo D con un "passo" della dimensione C. Assumiamo che DESCRIZIONE sia una lista degli argomenti di PROG, senza il primo argomento ovviamente, che sono le variabili da definire. Il valore di

(FOR A B C D DESCRIZIONE)

sarà allora una lista degli argomenti di PROG.

Esistono delle difficoltà nell'uso della funzione FOR*?*
Sì, c'è un problema molto importante nella costruzione della funzione FOR, per la precisione nella scelta dei nomi per le etichette X e Y. Come si fa a sapere che i simboli X e Y non sono già stati usati come etichette nel pezzo di programma rappresentato dalla DESCRIZIONE? Non lo si sa a priori. E allora, si può chiamare la funzione FOR più di una volta, anche molte volte, e poi combinare i brani più piccoli di programma in un brano più grande. Certo non vogliamo che le etichette si ripetano: la duplicazione delle etichette infatti può creare problemi.

Che cosa si può fare?
Si può usare la funzione GENSYM. Questa funzione genera un simbolo diverso da tutti i simboli generati fino a quel momento. GENSYM non ha argomenti e può essere usata tutte le volte che si vuole. Basta ricordare che genererà un nuovo simbolo ogni volta che viene chiamata.

Si può definire la funzione FOR *in termini di altre funzioni?*
Sì, si può scrivere il programma seguente:

```
DEFINE((
(FOR(LAMBDA(A B C D E) (PROG(F G H I)
(SETQ G (APPEND E (LIST
(LIST(QUOTE SETQ)
A (LIST(QUOTE PLUS)A C)))))
(SETQ H(GENSYM))
(SETQ I(GENSYM))
(SETQ G (APPEND G
(LIST(LIST(QUOTE GO)
H)I)))
(SETQ F(LIST (LIST(QUOTE
SETQ) A B)
H(LIST(QUOTE COND)
(LIST(LIST(QUOTE
GREATERP) A D)
(LIST(QUOTE GO)
I)))))
(RETURN(APPEND F G)) )))))
```

INPUT/OUTPUT

Ci sono enunciati per la lettura e la scrittura di dati in Lisp?
Sì, sono come tutti gli altri enunciati del linguaggio. Sono funzioni con argomenti come ogni altra funzione con argomenti. Ovviamente non è necessario usare enunciati di input/output per sfruttare il Lisp nella risoluzione di un problema: tutto quel che si deve fare è definire una funzione e poi chiamarla con gli argomenti opportuni, se necessari. Il Lisp

stampa automaticamente i risultati. Se non ci sono argomenti, basta chiamare la funzione entro parentesi.

Esistono funzioni speciali per l'input/output?
Sì, sono progettate per dare completezza al linguaggio e renderne più semplice l'utilizzazione nella programmazione.

Quali sono queste funzioni?
Sono READ, PRINT e TERPRI.

Che cos'è la funzione READ*?*
READ significa "leggere" ed è una funzione senza argomenti. Tutte le volte che viene chiamata la funzione READ, il sistema Lisp legge un'intera S-espressione. In altre parole, legge un carattere non vuoto. Se questo carattere non è una parentesi aperta, il sistema assume che sia l'inizio di un atomo. Il sistema operativo del Lisp poi continua a leggere finché non raggiunge un carattere vuoto (uno spazio) o un altro carattere di separazione che segnala la fine dell'atomo. Se il primo carattere letto era invece una parentesi aperta, il Lisp continua a leggere finché non ha raggiunto la fine della S-espressione. Ovviamente conta i livelli di parentesi per assicurarsi di non leggere troppo poco o troppo.

Questo sistema presenta svantaggi?
Sfortunatamente, c'è una situazione che crea uno svantaggio serio. Se c'è un solo errore su una linea in ingresso, da un terminale o da un lettore di schede, l'errore può inibire un'ulteriore lettura di dati o di linee in ingresso. Questo si verifica in particolare se l'errore è una parentesi chiusa mancante in un punto qualunque di una linea in ingresso. Se non ci sono errori, invece, il valore della funzione READ è la S-espressione risultante.

Che cosa c'è da dire sulla funzione PRINT*?*
La funzione PRINT ha un solo argomento, e determina la stampa della S-espressione di tale argomento. Se si usa la funzione PRINT al livello 0, la S-espressione verrà stampata due volte.

E perché due volte?
Perché le funzioni PRINT e READ, oltre a essere usate per ottenere un effetto come la stampa o la lettura di dati, hanno anche valori. Il valore è quello dell'argomento della funzione. Pertanto, se si pone A uguale a (6 7 8 9) e poi si scrive:

```
PRINT(A)
```

al livello 0, la prima cosa che succede è la stampa di A — come ci si aspettava — perché PRINT è la funzione di stampa. Poi però il Lisp si comporta come sempre al livello 0: stampa il valore della funzione chiamata. Il problema svanisce quando PRINT viene usata all'interno di un programma.

Perché si usa PRINT *al posto di* RETURN*?*
Di solito la funzione PRINT viene usata per mandare in uscita un valore temporaneo, per esempio a metà di un calcolo.

C'è altro da dire su PRINT*?*
Se si lavora su un sistema in time sharing, di solito è necessaria solo la funzione PRINT per mandare in uscita le informazioni sul terminale dell'utente. Tuttavia, se si usa una stampante a riga, viene stampata una riga completa alla volta. Usando PRINT durante l'esecuzione di un programma, gran parte dell'uscita apparirà molto dopo che la funzione PRINT è stata eseguita, perché in genere le stampanti a riga richiedono il riempimento di una memoria di transito prima di stampare una riga. Questa memoria di transito è una piccola memoria all'interno della stampante che immagazzina una riga intera di caratteri per la stampante. Il problema è che dopo la conclusione del programma di solito in questa memoria di transito si trovano ancora delle informazioni in attesa di stampa.

E come si fa per stampare queste informazioni?
Si usa la funzione TERPRI. TERPRI è un acronimo per "terminate printing", ovvero "concludi la stampa". TERPRI viene usata soprattutto alla fine di un programma in Lisp, ma nulla impedisce di usarla anche altrove.

In quali altre situazioni si può usare TERPRI*?*
In genere, viene usata prima di una funzione PRINT: in questo modo si ha la sicurezza che venga stampata una nuova riga.

ANCORA UN RIPASSO

CASA *è un atomo?*
Sì, perché è una stringa di caratteri.

TOPI *è un atomo?*
Sì.

6789 *è un atomo?*
Sì, perché è una stringa di caratteri.

45TRE56 *è un atomo?*
Sì.

Y *è un atomo?*
Sì.

6 *è un atomo?*
Sì.

(TOPI) *è una lista?*
Sì, perché (TOPI) è un atomo racchiuso fra parentesi.

(I TOPI SONO SIMPATICI) *è una lista?*
Sì, perché è un insieme di atomi racchiusi fra parentesi.

(I TOPI SONO) SIMPATICI *è una lista?*
No, perché si tratta solo di due S-espressioni non racchiuse complessivamente fra parentesi. La prima S-espressione è una lista, la seconda è un atomo.

((I TOPI SONO) SIMPATICI) *è una lista?*
Sì.

TOPI *è una S-espressione?*
Sì, tutti gli atomi sono S-espressioni.

(TOPI) *è una S-espressione?*
Sì, tutte le liste sono S-espressioni.

(QUESTA SEMBRA UNA LISTA) *è una lista?*
Sì, perché è un insieme di S-espressioni racchiuse fra parentesi.

((4)7) *è un esempio di S-espressione?*
Sì.

Quante S-espressioni compaiono nella lista

((I TOPI) SONO)((DIVERTENTI)(DA GUARDARE))((DALLA FINESTRA))*?*

Tre, e precisamente:

((I TOPI) SONO), ((DIVERTENTI)(DA GUARDARE)) e
((DALLA FINESTRA)).

Questa è una lista: ()*?*
Sì, è la lista vuota.

(()()()()) *è una lista?*
Sì.

Che cos'è il CAR *di* (A B C)*?*
A.

Che cos'è il CAR *di* ((5 6)8 9)*?*
(5 6).

Che cos'è il CAR *di* TOPI*?*
Non c'è risposta, non si può chiedere il CAR di un atomo.

Che cos'è il CAR *di* ()*?*
Non c'è risposta, non si può chiedere il CAR della lista vuota.

Che cos'è il CDR *di* (7 8 9)*?*
(8 9).

Che cos'è il CDR *di* (A S(U I))*?*
(S(U I)).

Che cos'è il CDR *di* ((W E R)Y U I)*?*
(Y U I).

Che cos'è il CDR *di* TOPI*?*
Non si può chiedere il CDR di un atomo, pertanto non c'è risposta.

Che cos'è il CDR *di* ()*?*
Non c'è risposta, non si può chiedere il CDR della lista vuota.

Che cos'è il CDR *di* (A)*?*
NIL.

Che cos'è il CADR *di* ((T)(Y U)(H))*?*
(Y U).

Come si esprime il CAR *del* CDR*?*
CADR.

Come si esprime il CDR *del* CDR*?*
CDDR.

Come si esprime il CDR *del* CDR *del* CDR*?*
CDDDR.

Come si esprime il CAR *del* CDR *del* CDR*?*
CADDR.

Qual è il valore di (APPEND A B) *se* A *è* (7 8) *e* B *è* (5 6)*?*
(7 8 5 6).

Qual è il valore di (APPEND A B) *se* A *è* (4 5 6) *e* B *è* (9)*?*
(4 5 6 9).

Qual è il valore di (CONS A B) *se* A *è* (56) *e* B *è* (7 8)*?*
((56)7 8).

Qual è il valore di (CONS A B) *se* A *è* (5 7) *e* B *è* ((4 6))*?*
((5 7)(4 6)).

Qual è il valore di (ATOM A) *se* A *è* 7*?*
T.

Qual è il valore di (ATOM A) *se* A *è* (5 6 7)*?*
NIL.

Qual è il valore di (NULL A) *se* A *è* 9*?*
NIL.

Qual è il valore di (NULL A) *se* A *è* (6 7 8)*?*
NIL.

Qual è il valore di (NULL A) *se* A *è* NIL*?*
T.

Qual è il valore di (NULL(CDR A)) *se* A *è* (3)*?*
T.

Qual è il valore di (EQUAL A B) *se* A *è* (7) *e* B *è* (7)*?*
T.

Qual è il valore di (EQUAL A B) *se* A *è* (TIMES 4 5) *e* B *è* 20*?*
NIL.

Se A *è* 5 *e* B *è* (2 3 4 5 6), *che cos'è* (MEMBER A B)*?*
T.

Se A *è* 7 *e* B *è* (8 5 9), *che cos'è* (MEMBER A B)*?*
NIL.

Se A *è* 3 *e* B *è* (4445), *che cos'è* (MEMBER A B)*?*
NIL.

Qual è il valore di (OR T T)*?*
T.

Qual è il valore di (OR T NIL)*?*
T.

Qual è il valore di (OR NIL T)*?*
T.

Qual è il valore di (OR NIL NIL)*?*
NIL.

Qual è il valore di (AND NIL T)*?*
NIL.

Qual è il valore di (AND T NIL)*?*
NIL.

Qual è il valore di (AND NIL NIL)*?*
NIL.

Qual è il valore di (AND T T)*?*
T.

Qual è il valore di (NUMBERP A), *se* A *è* 6*?*
T.

Qual è il valore di (NUMBERP A), *se* A *è* T *(il simbolo)?*
NIL.

Qual è il valore di (NUMBERP A), *se* A *è* (5 6 7)*?*
NIL.

Qual è il valore di (FIXP A), *se* A *è* (5 6 7)*?*
NIL.

Qual è il valore di (FIXP A), *se* A *è* 8.9*?*
NIL.

Qual è il valore di (FIXP A), *se* A *è* 9*?*
T.

Qual è il valore di (FLOATP A), *se* A *è* 9*?*
NIL.

Qual è il valore di (FLOATP A), se A *è* 8.9*?*
T.

Se A *è* (5 6 89 9.8 67 5.9) *qual è il valore di* (COUNTFIXFLOAT A)*?*
(4 2).

Se A *è* (TIMES 6 7), *che cos'è* (EVAL A)*?*
42.

Se A *è* (LIST 678 5 89), *che cos'è* (EVAL A)*?*
(678 5 89).

Se A *è* (QUOTE TOPI), *che cos'è* (EVAL A)*?*
TOPI.

CAPITOLO CINQUE

Lisp e intelligenza artificiale

ANCORA LISP

Che cosa manca?
In effetti, abbiamo visto quasi tutto, del Lisp. Ci sono ancora altri elementi, ma, come al solito con i linguaggi di programmazione, insorge il problema dei dialetti. Ogni sistema di calcolo può disporre di una versione leggermente diversa di Lisp, ma è raro che le differenze siano molto significative. Il Lisp è uno dei pochi linguaggi che risultano relativamente standardizzati da macchina a macchina. Anche nelle versioni standard, però, c'è ancora qualche punto cui dobbiamo accennare.

Che cosa fa la funzione RECIP*?*
La funzione RECIP dà come valore il reciproco dell'argomento. Quindi:

(RECIP 5) è 0.2
(RECIP 3) è 0.333333
(RECIP 7) è 0.142857
(RECIP 2) è 0.5.

Che cosa fa la funzione FLOAT*?*
La funzione FLOAT trasforma un numero intero in un numero reale. Un numero reale è un numero in virgola mobile. Questo significa che può comprendere dei decimali. Quindi:

```
(FLOAT 5) è 5.0
(FLOAT 67) è 67.0
(FLOAT 3) è 3.0
(FLOAT 100) è 100.0.
```

Che cos'è la funzione ENTIER *e che cosa fa?*
La funzione ENTIER trasforma un numero in virgola mobile in un intero. Quindi:

```
(ENTIER 7.0) è 7
(ENTIER 6.0) è 6
(ENTIER 34.0) è 34
(ENTIER 78.0) è 78.
```

Che cosa fa EXPT *in Lisp?*
EXPT è la funzione esponenziale. Il suo valore è il primo argomento elevato alla potenza indicata dal secondo argomento:

```
(EXPT 2 5) è 32
(EXPT 4 7) è 16384
(EXPT 3 8) è 6561
(EXPT 7 3) è 343.
```

Che cos'è la funzione MAX*?*
La funzione MAX (da *maximum*, massimo) dà il valore più grande nella stringa dei suoi argomenti:

```
(MAX 5 87 3 45 92) è 92
(MAX 654 34 2 897) è 897
(MAX 567 76 234 56) è 567
(MAX 1024 678 987) è 1024.
```

Che cos'è la funzione MIN*?*
È l'opposto della funzione MAX. Il valore che dà è il minimo fra tutti quelli presenti nella lista dei suoi argomenti:

```
(MIN 56 4 78 43) è 4
(MIN 78 654 34) è 34
(MIN 567 1098 3456) è 567
(MIN 78 6 345 1) è 1.
```

Esistono in Lisp funzioni per l'incremento e il decremento?
Sì, sono ADD1 e SUB1. ADD1 somma 1 al suo argomento, SUB1 sottrae 1 al suo argomento. Ecco qualche esempio:

(SUB1 45) è 44
(SUB1 23) è 22
(SUB1 1025) è 1024
(SUB1 1) è 0
(ADD1 56) è 57
(ADD1 0) è 1
(ADD1 1067) è 1068
(ADD1 457) è 458.

Come si può determinare se un numero è pari o dispari?
Si usa la funzione EVENP che dà T se il suo argomento è pari e NIL se è dispari:

(EVENP 6) è T
(EVENP 77) è NIL
(EVENP 1024) è T
(EVENP 1023) è NIL.

In Lisp una funzione può definire un'altra funzione e poi usare quella funzione che ha appena definito?
Sì, di solito lo si fa usando la funzione DEFINE a un livello interno, nel programma o nella routine. In altre parole, la descrizione della funzione data (secondo argomento di LAMBDA) può a sua volta contenere una funzione DEFINE.
Se una particolare routine o un particolare programma contiene un numero elevato di nomi, possono verificarsi problemi seri. L'interprete o il compilatore Lisp possono risultare sovraccaricati. Sono state sviluppate varie tecniche per risolvere il problema. Si può far passare il nome della funzione interna come un parametro:

(A B C D(QUOTE Q))

La funzione è A con i valori (parametri B, C e D) che nel corso dell'esecuzione del programma definiranno una funzione interna chiamata Q. Alcuni sistemi Lisp usano la parola FUNCTION invece di QUOTE. E ovviamente si può usare la funzione LAMBDA in qualunque punto di un programma.

Che cos'è la funzione LABEL *in Lisp?*
La funzione LABEL ("etichetta") ha due argomenti. Il primo è il nome di una funzione, mentre il secondo è una descrizione della funzione così etichettata. Tipicamente, il secondo argomento conterrà una espressione LAMBDA.

Che cos'è la funzione FUNCALL*?*
La funzione FUNCALL consente il calcolo di nomi o descrizioni di funzioni. Il primo argomento viene usato per calcolare un nome di funzione o una espressione (descrizione) LAMBDA. Poi la funzione risultante viene applicata al resto degli argomenti (che saranno tanti quanti necessari per la funzione così definita).

C'è ancora qualcosa di importante da dire a proposito di QUOTE*?*
Quando il codice Lisp viene letto nella memoria centrale da un terminale o da un dispositivo di memoria di massa, le virgolette singole (se sono state usate) vengono tradotte nelle applicazioni formali della funzione QUOTE. È un passo indispensabile, perché l'interprete Lisp richiede che tutti i programmi e i dati siano in forma di S-espressioni.

Che cosa sono le virgolette singole?
Le virgolette singole sostituiscono la parola QUOTE e le parentesi che l'accompagnano. Così, 'S-espressione equivale a (QUOTE S-espressione).

Il Lisp può essere un interprete, ma può anche essere un compilatore. Come è possibile?
Quasi tutti i sistemi Lisp sono solo interpreti, ma hanno una funzione chiamata COMPILE.

E che cosa fa questa funzione?
La funzione COMPILE consente all'utente di compilare una funzione Lisp. Compilando una funzione si può aumentare nettamente la velocità di esecuzione. La compilazione della funzione trasforma le parole in una serie di istruzioni nel linguaggio macchina dello specifico computer utilizzato.

C'è qualche differenza significativa fra il Lisp compilato e altri linguaggi compilati?
Sì, altri linguaggi richiedono solo la presenza di una certa porzione del linguaggio. Per esempio, in Fortran deve essere presente il modulo RUN-TIME, che contiene subroutine utili come le routine di input-output. Il codice oggetto compilato del Lisp richiede tutte le risorse del sistema Lisp per le subroutine. Pertanto in memoria deve restare tutto il sistema Lisp. Lo stesso succede per i sistemi WATFOR e WATFIV.

Che cos'è la funzione SPECIAL*?*
La funzione SPECIAL ha solo un argomento. È una lista di variabili (cioé dei nomi delle variabili). La funzione assicura che le variabili conservino il loro valore da una utilizzazione del programma all'altra. È una funzione simile alla OWN dell'Algol.

Che cos'è UNSPECIAL?
Come lascia pensare il nome, è l'inversa di SPECIAL: dichiara che le variabili nella lista che costituisce il suo argomento non debbono più essere considerate "speciali".

Che cos'è la "garbage collection" in Lisp?
Il sistema Lisp usa grandi quantità di memoria, per conservare tutti i puntatori di lista e le liste circolari. Queste ultime sono liste che non sono S-espressioni: spesso non hanno parentesi chiusa, ma sono molto utili. La "garbage collection", cioè la "raccolta dei rifiuti" in Lisp è un procedimento di ricerca delle parole disponibili come memoria libera, che vengono raccolte in una lista. In questo modo il sistema gira senza riempire tutta la memoria con liste non più necessarie. La routine di "raccolta dei rifiuti" inserita nel Lisp deve passare in rassegna tutte le liste presenti in memoria e deve determinare quali non sono più necessarie.

INTELLIGENZA ARTIFICIALE

Il Lisp, si è detto, è strettamente legato alle ricerche sull'intelligenza artificiale, al punto da risultare, agli occhi di molti, del tutto inscindibile da questo settore della scienza: c'è anche chi sostiene che, senza il Lisp, l'intelligenza artificiale non esisterebbe nemmeno, come seria disciplina scientifica. Due parole in merito, dunque, sono d'obbligo.
L'intelligenza artificiale studia come si possano creare programmi di calcolatore che simulino i processi che sembrano richiedere l'intelligenza di un uomo. I programmi di intelligenza artificiale sono usati, per esempio:

- negli uffici, per distribuire nel tempo il carico di lavoro, per organizzare la documentazione del personale, preparare rapporti e usare le informazioni disponibili per formulare decisioni;
- in agricoltura e nell'allevamento, per fornire informazioni sui raccolti e sugli animali nocivi, a partire dai dati raccolti;
- nella produzione industriale, per il controllo di apparecchiature e di conseguenza per la diminuzione degli errori;
- negli ospedali, per correlare i sintomi con i disturbi mentali e fisici.

I computer possono anche risolvere test di intelligenza. Sono stati formulati programmi che consentono a un computer di analizzare un problema e arrivare alla soluzione corretta.
Un programma può "decidere" che una struttura "a T" è una costruzione che richiede il posizionamento di un blocco in cima a un altro, centrato. Per giungere a un risultato del genere è necessario disporre di

strumenti per la soluzione di problemi di identificazione e rappresentazione di immagini (fig. 5.1).
È possibile "insegnare" a un programma le differenze fra linee, ombre e chiaroscuri. Un programma può "vedere" una scena grazie a un sistema di ripresa fotografica a codifica digitale, e riconoscere poi i componenti della scena (fig. 5.2).

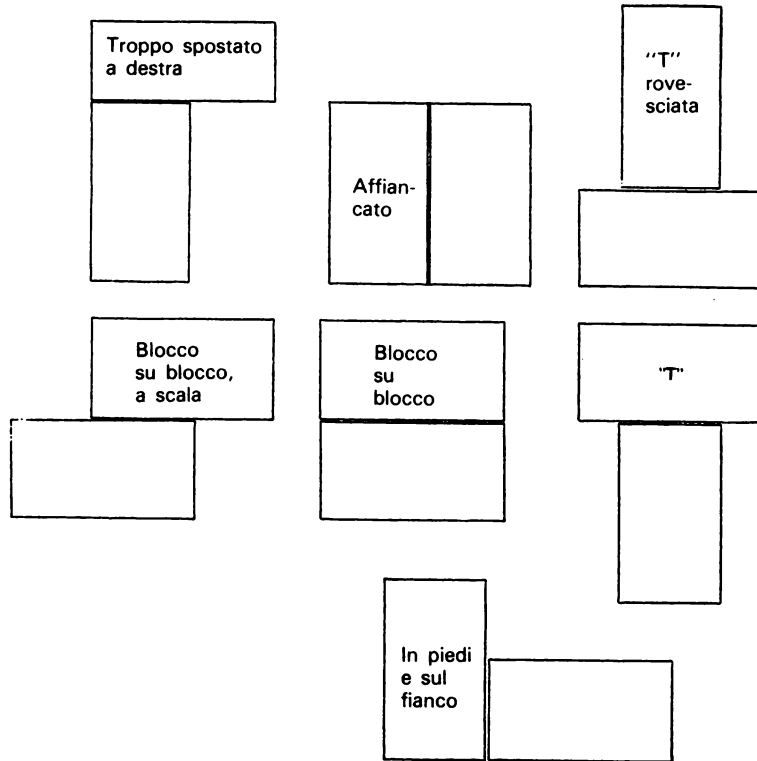



Fig. 5.1.

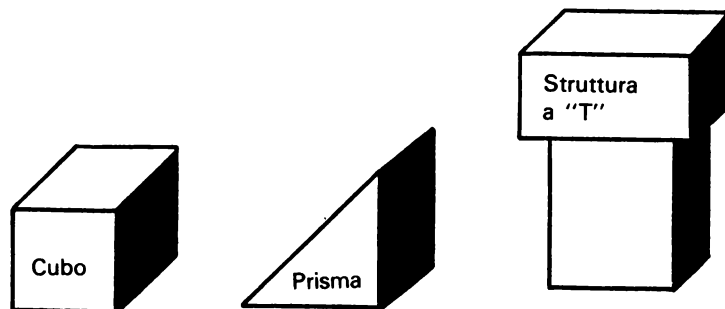



Fig. 5.2. Tipicamente, le scene "viste dal programma" sono piene di ombre, linee, confini, curve e fratture.

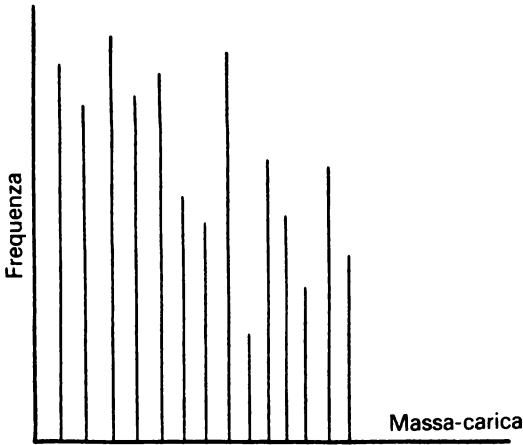



Fig. 5.3.

Si possono progettare programmi che correlano i dati forniti da uno spettrometro di massa per determinare la presenza di materiali ed elementi diversi. Con una macchina "intelligente" il margine di errore è notevolmente inferiore, e non è richiesto l'intervento dell'uomo. I programmi debbono essere in grado di apprendere e di automodificarsi (fig. 5.4).

Fig. 5.4.

Un programma può fare riferimento a un insieme o a insiemi di parametri per determinare se l'oggetto "visto" è l'oggetto desiderato, mettendo in corrispondenza biunivoca i parametri dell'oggetto visto e quelli dell'oggetto "ideale" (fig. 5.5).

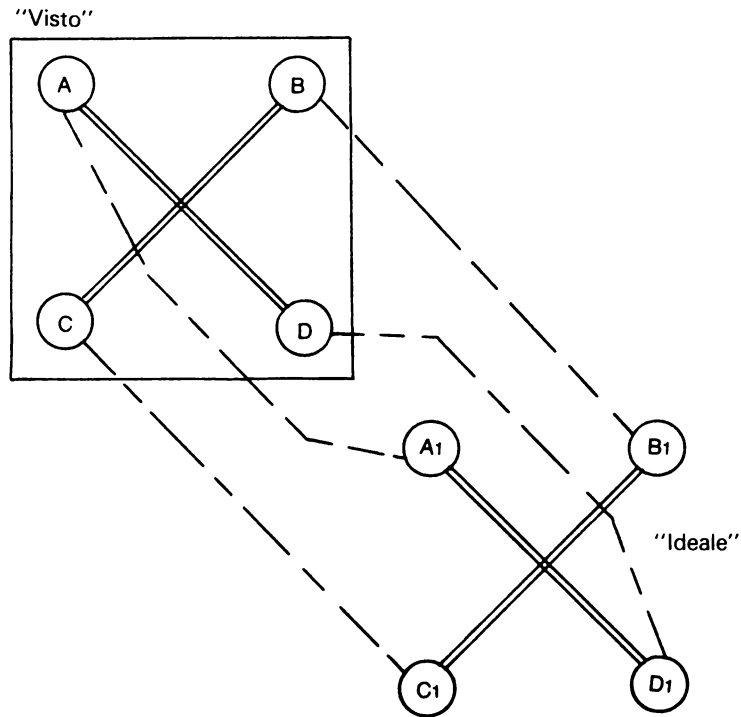



Fig. 5.5.

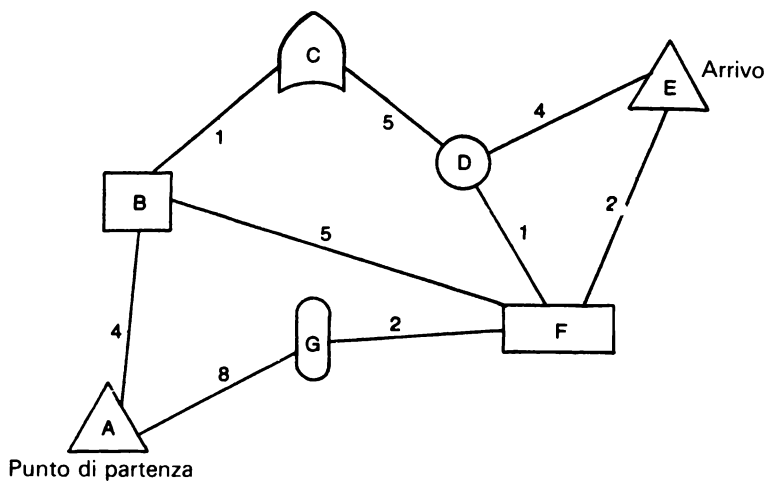



Fig. 5.6.

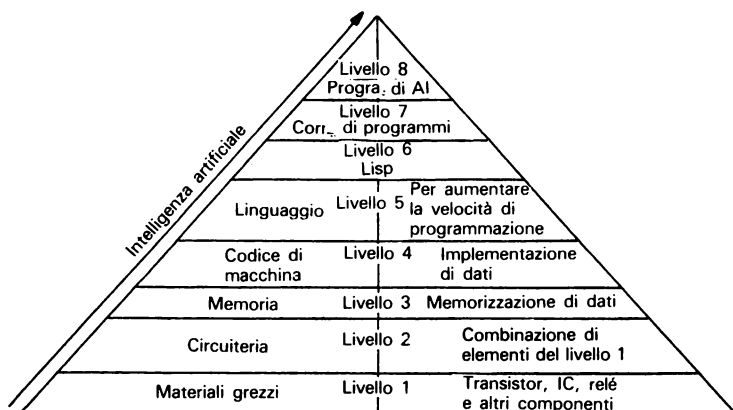



Fig. 5.7.

Un problema di ricerca richiede che il programma analizzi tutte le informazioni che gli sono fornite. Il problema è quello di cercare o trovare il percorso migliore dal punto *A* al punto *E* in termini di distanza minima (fig. 5.6). I numeri si riferiscono a unità di distanza (unità relative). È ovvio che il percorso migliore va da *A* a *B* , poi a *F* e infine a *E*.
I programmi di intelligenza artificiale, come ogni altra struttura molto raffinata, dipendono da fondamenta ampie e solide (fig. 5.7).

CAPITOLO SEI

Programmi ed esempi

Questo capitolo contiene programmi ed esempi in Lisp. Va notato che in certe routine non è stata usata la funzione LAMBDA. Taluni sistemi Lisp consentono l'esecuzione di una definizione senza il ricorso al LAMBDA: in questo caso, gli argomenti della funzione che viene definita sono racchiusi entro le stesse parentesi con il nome della funzione.

AGG

Obiettivo

Usare una lista come una "base di dati" e aggiungere (AGG) informazioni a questa base di dati. Il nome della lista è DATA e I la nuova informazione.

Programma

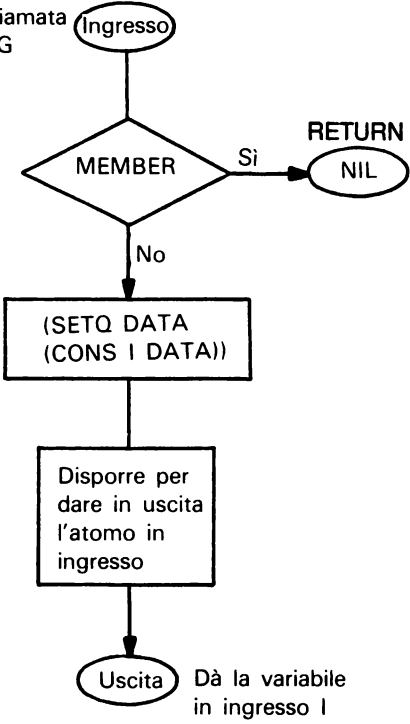
```
(DEFINE
(AGG I)
(COND ((MEMBER
(I DATA)
NIL)
(T
(SETQ DATA
(CONS I DATA))
)))
```

Si noti che questa routine controlla se la nuova informazione I è già presente nella lista, prima di aggiungerla ad essa. La funzione dà in uscita NIL se l'informazione è già presente, mentre dà l'informazione (I) se questa non era presente ed è stata aggiunta alla "base di dati".

Esecuzione campione

Assumiamo che la lista DATA contenga gli elementi TOPI CANE LEONE.

```
    (SETQ 'DATA '(TOPI GATTO LEONE))
    (SETQ I 'TOPO)
    (AGG I)
    TOPO

******************************

    (SETQ I 'TOPI)
    (AGG I)
    NIL

******************************

    (SETQ I 'CANE)
    (AGG I)
    CANE

******************************

    (SETQ I 'GATTO)
    (AGG I)
    NIL

******************************

    END
```

Ingresso dalla chiamata
alla funzione AGG
Ingresso
MEMBER
Sì
RETURN
NIL
No
(SETQ DATA
(CONS I DATA))
Disporre per
dare in uscita
l'atomo in
ingresso
Uscita
Dà la variabile
in ingresso I

MOMENTO D'INERZIA DI UN ANELLO: MIA

Obiettivo

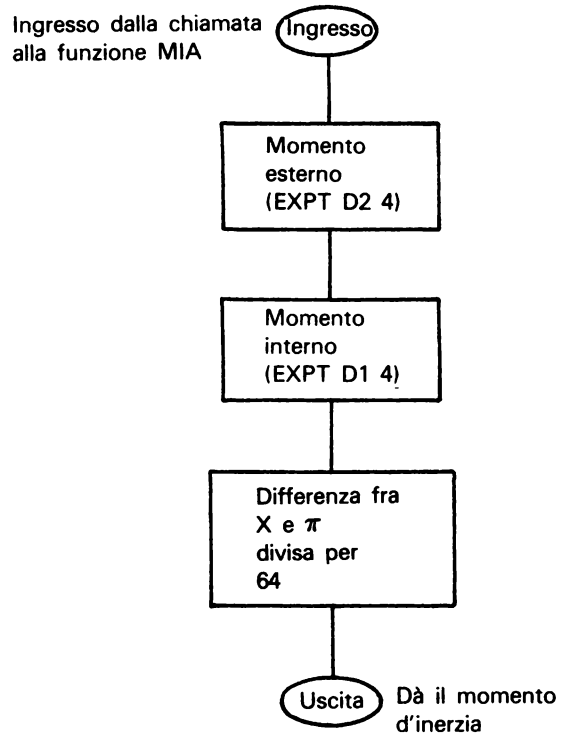
Trovare il momento d'inerzia di un anello.

Programma

```
(DEFINE
(MIA D1 D2)
(QUOTIENT
(TIMES
(DIFFERENCE
(EXPT D2 4)
(EXPT D1 4))
3.14159)
64)
)
```

Esecuzione campione

```
   (MIA 1 2)
   0.7363

******************************

   (MIA 3 4)
   8.590

******************************

   (MIA 5 6)
   32.93

******************************

   (MIA 7 8)
   83.20

******************************

   END
```

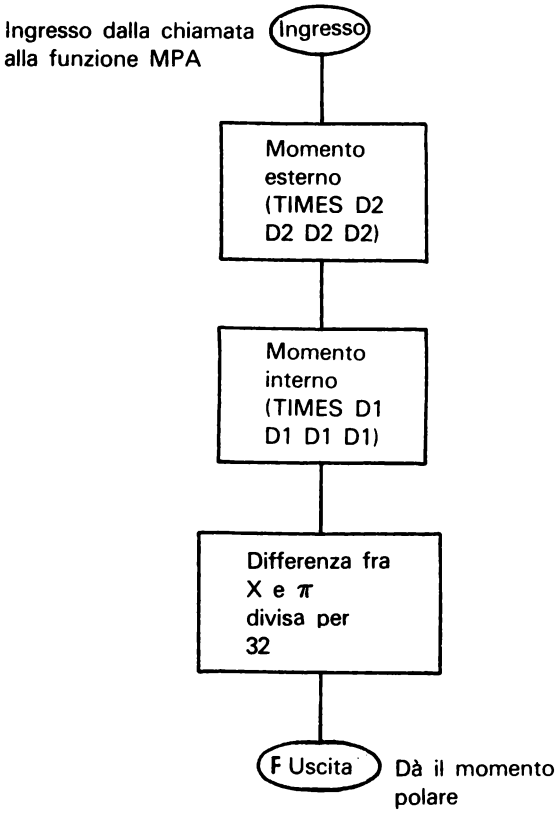
Diagramma di flusso: momento di inerzia di un anello

MOMENTO POLARE DI UN ANELLO: MPA

Programma

```
(DEFINE
(MPA D2 D1)
(QUOTIENT
(TIMES
(DIFFERENCE
(TIMES D2 D2 D2 D2)
(TIMES D1 D1 D1 D1))
3.14159)
32)
)
```

Esecuzione campione

```
   (MPA 2 1)
   1.4726

*******************************

   (MPA 4 3)
   17.1806

*******************************

   (MPA 8 7)
   166.406

*******************************

   (MPA 10 9)
   337.623

*******************************

   END
```

Diagramma di flusso: momento polare di un anello

CONTROLLO DEGLI ATOMI: LIS

Obiettivo

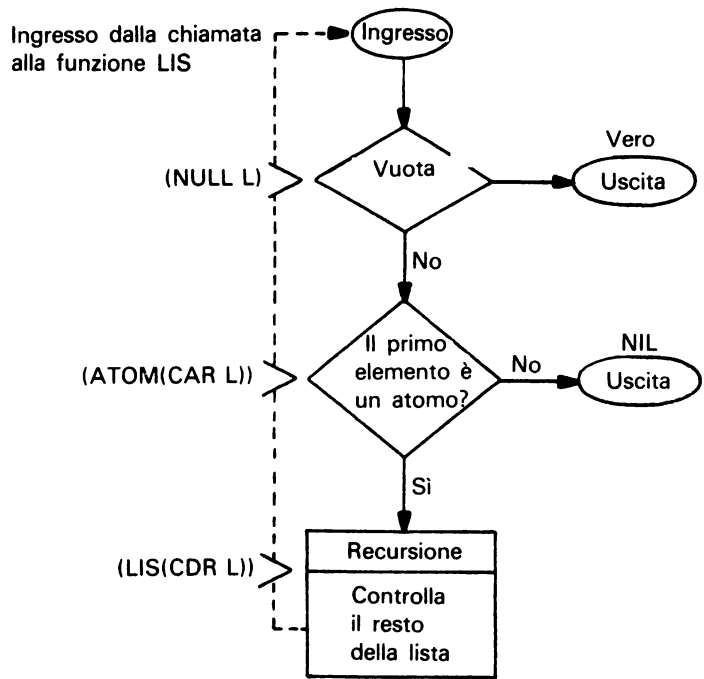
Controllare una lista per determinare se contiene solo atomi e nessuna sottolista.

Programma

```
(DEFINE
(LIS L)
(COND
((NULL L)
T)
((ATOM (CAR L))
(LIS (CDR L)))
(T NIL'
)))))
```

Esecuzione campione

```
   (LIS '(GIORGIO STA BENE))
   T

******************************

   (LIS '(GIORGIO (STA) BENE))
   NIL

******************************

   (LIS '(QUESTO RISULTA (UN) BUON TEST))
   NIL

******************************

   (LIS '(GLI ELEFANTI SONO SEMPRE AFFAMATI))
   T

******************************

   (LIS '())
   T

******************************

   (LIS '((I TOPI) VANNO (BENE) COL (RISO))
   NIL

******************************
```

Diagramma di flusso: controllo degli atomi

(Nota: la riga tratteggiata si riferisce al percorso ricorsivo)

ATOMO ELEMENTO DI UNA LISTA: MEM

Obiettivo

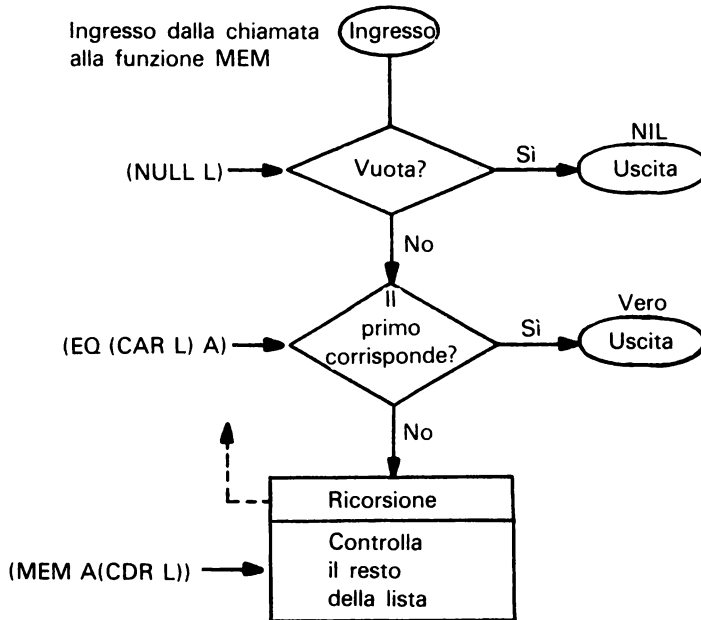
Controllare se il primo argomento, un atomo, è un membro del secondo argomento, che è una lista.

Programma

```
(DEFINE
(MEM A L)
(COND
((NULL L)
NIL)
((EQ (CAR L) A)
T)
(T MEM A (CDR L)))
))
```

Esecuzione campione

```
   (MEM'STA'(GIORGIO STA BENE))
   T

*******************************

   (MEM'CASA'())
   NIL

*******************************

   (MEM'A'(G Y E A D Y))
   T

*******************************

   (MEM'F'(I J U F O))
   T

*******************************

   (MEM'ROSSO'(ROSA VERDE ARANCIONE))
   NIL

*******************************

   END
```

Diagramma di flusso: atomo elemento di una lista

(Nota: la ricorsione fa tornare al punto iniziale della chiamata della routine, con nuove variabili. La linea tratteggiata si riferisce al percorso ricorsivo)

MOMENTO POLARE CIRCOLARE: MPC

Obiettivo

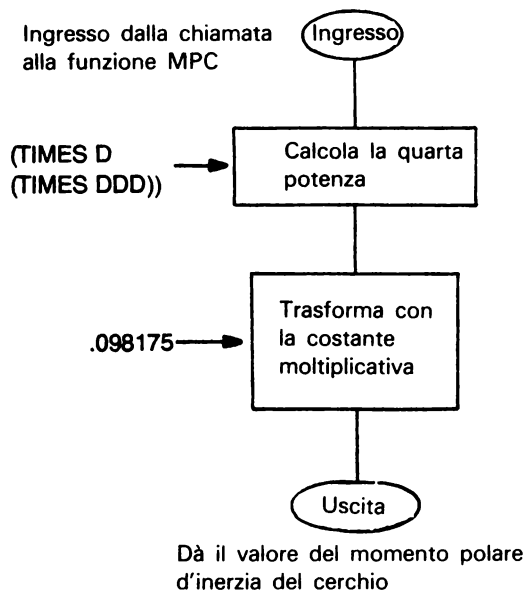
Trovare il momento polare d'inerzia di un cerchio.

Programma

```
(DEFINE
(MPC D)
(TIMES
(TIMES D
(TIMES D D D))
.098175)
```

Esecuzione campione

```
   (SETQ D 1)
   (MPC D)
   .098175

*******************************

   (SETQ D 2)
   (MPC D)
   1.5708

*******************************

   (MPC 3)
   7.95215

*******************************

   (MPC 4)
   25.133

*******************************

   END
```

Diagramma di flusso: momento polare circolare

ADDIZIONE COMPLESSA: COMPLUS

Obiettivo

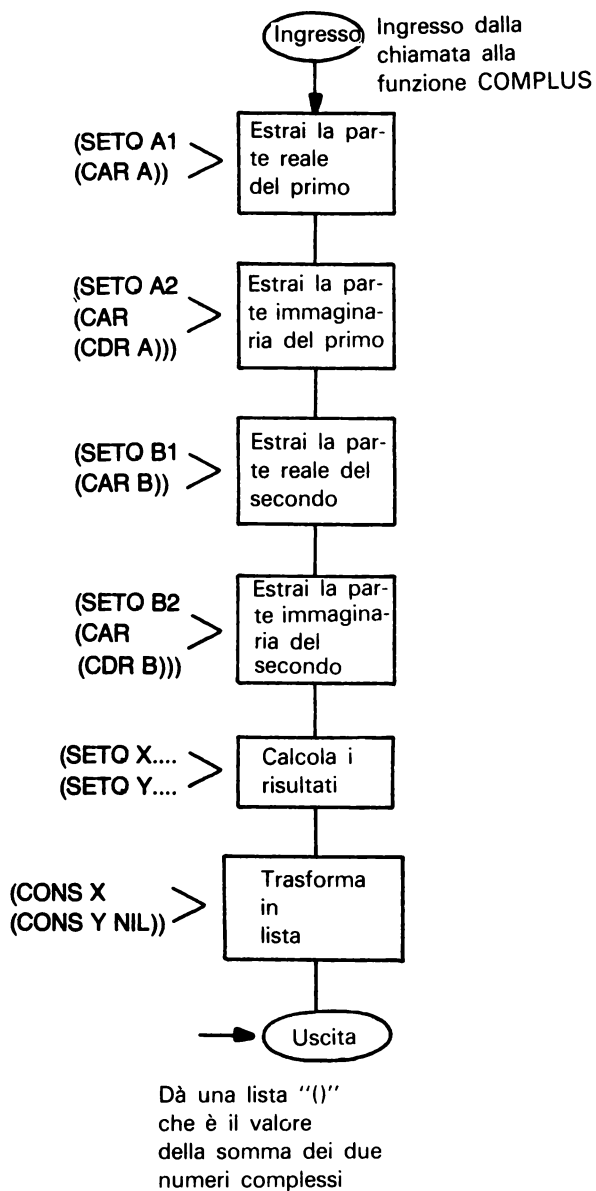
Trovare la somma di due numeri complessi.

Programma

```
(DEFINE
(COMPLUS A B)
(SETQ A1
(CAR A))
(SETQ A
(CAR (CDR A)))
(SETQ B1
(CAR B))
(SETQ B2
(CAR (CDR B)))
(SETQ X (PLUS A1 B1))
(SETQ Y (PLUS A2 B2))
(CONS X
(CONS Y NIL))
)
```

Esecuzione campione

```
   (COMPLUS '(1 0) '(2 0)
   (3 0)

*******************************

   (COMPLUS '(1 2) '(3 4))
   (4 6)

*******************************

   END
```

Diagramma di flusso: addizione complessa

CONIUGATO COMPLESSO: CCON

Obiettivo

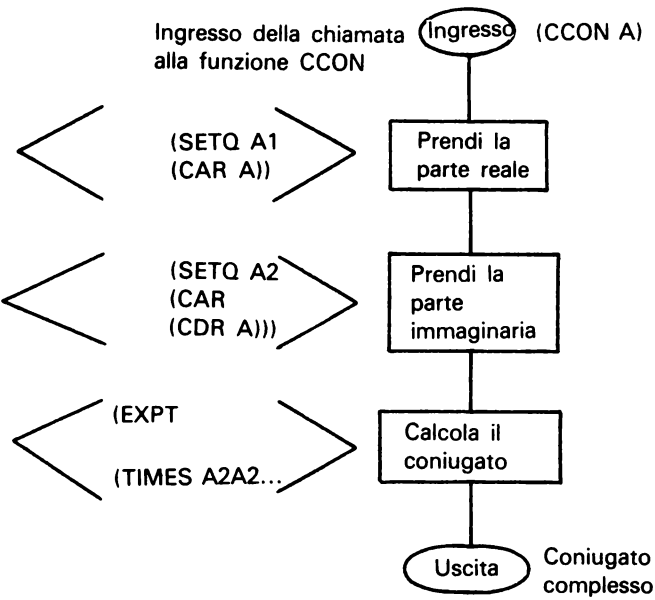
Trovare il valore assoluto di un numero complesso.

Programma

```
(DEFINE
(CCON A)
(SETQ A1
(CAR A))
(SETQ A2
(CAR
(CDR A)))
EXPT
(PLUS
(TIMES A1 A1)
(TIMES A2 A2)) .5)
```

Esecuzione campione

```
 (CCON '(1 0))
 1

********************************

 (CCON '(1 1))
 1.414

********************************

 (CCON '(2 3))
 3.605

********************************

 (CCON '(4 5))
 6.403

********************************

 END
```

Diagramma di flusso: coniugato complesso

Dà il valore assoluto del numero complesso inserito

DIVISIONE COMPLESSA: COMQUO

Obiettivo

Trovare il quoziente di due numeri complessi.

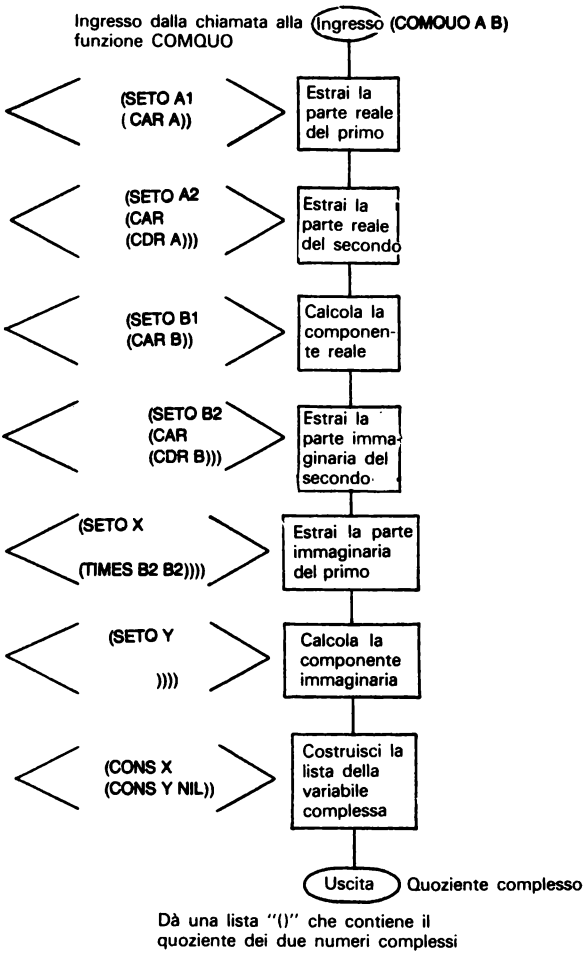
Programma

```
(DEFINE
(COMQUO A B)
(SETQ A1
(CAR A))
(SETQ A2)
(CAR
(CDR A)))
(SETQ B1
(CAR B))
(SETQ B2
(CAR
(CDR B)))
(SETQ X
(QUOTIENT
(PLUS
(TIMES A1 A2)
(TIMES B1 B2))
(PLUS
(TIMES A2 A2)
(TIMES B2 B2))))
(SETQ Y
(QUOTIENT
(DIFFERENCE
(TIMES A2 B1)
(TIMES A1 B2))
(PLUS
(TIMES A2 A2)
(TIMES B2 B2))))
(CONS X
(CONS Y NIL))
)
```

Esecuzione campione

```
   (COMQUO '(2 0) '(1 0))
   (2 0)

******************************

   (COMQUO '(1 1) '(2 1))
   (.6 .2)

******************************
```

```
(COMQUO '(12 15) '(27 32))
(.46 .11)

*********************************

END
```

Diagramma di flusso: divisione complessa

MOLTIPLICAZIONE COMPLESSA. COMTIMES

Obiettivo

Trovare il prodotto di due numeri complessi.

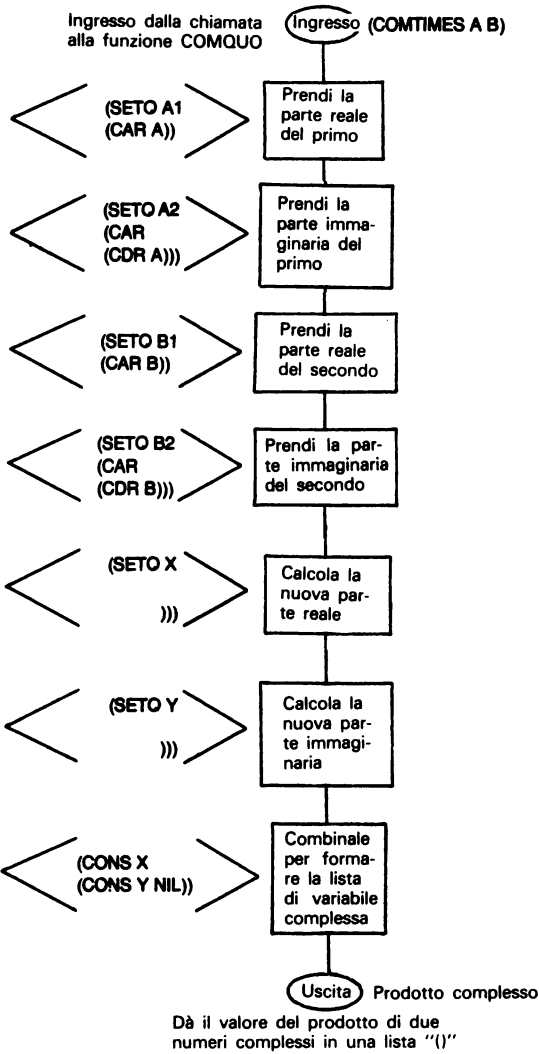
Programma

```
(DEFINE
(COMTIMES A B)
(SETQ A1
(CAR A))
(SETQ A2
(CAR
(CDR A)))
(SETQ B1
(CAR B))
(SETQ B2
(CAR
(CDR B)))
(SETQ X
(DIFFERENCE
(TIMES A1 A2)
(TIMES B1 B2)))
(SETQ Y
(PLUS
(TIMES A1 B2)
(TIMES A2 B1)))
(CONS X
(CONS Y NIL))
)
```

Esecuzione campione

```
    (SETQ A '(1 0))
    (SETQ B '(2 0))
    (COMTIMES A B)
    (2 0)

********************************

    (SETQ A '(1 1))
    (SETQ B '(2 1))
    (COMTIMES A B)
    (1 3)

********************************

    (COMTIMES '(1 1)'(4 7)
    (-3 11)

********************************
```

```
(COMTIMES '(12 15)'(27 32))
(-156 789)

********************************

END
```

Diagramma di flusso: moltiplicazione complessa

RECIPROCO COMPLESSO: CREC

Obiettivo

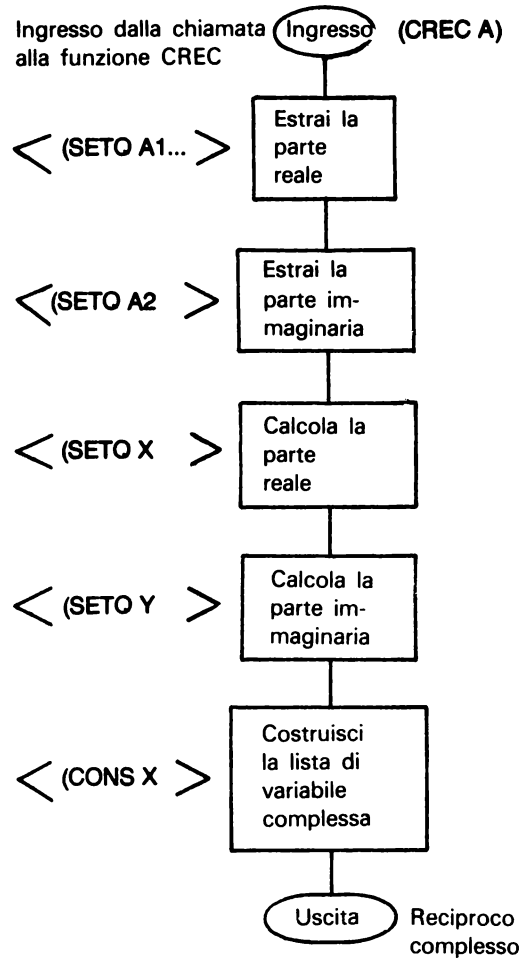
Trovare il reciproco di un numero complesso.

Programma

```
(DEFINE
(CREC A)
(SETQ A1 (CAR A))
(SETQ A2 (CAR (CDR A)))
(SETQ X (QUOTIENT A1 (PLUS
(TIMES A1 A1)
(TIMES B1 B1))))
(SETQ Y (QUOTIENT (MINUS B1)
(PLUS
(TIMES A1 A1)
(TIMES B1 B1))))
(CONS X (CONS Y NIL))
)
```

Esecuzione campione

```
    (CREC '(1 0))
    (1 0)

*******************************

    (CREC '(1 1))
    (.5 -.5)

*******************************

    (CREC '(2 3))
    (.15 -.23)

*******************************

    END
```

Diagramma di flusso: reciproco complesso

Dà il reciproco complesso del numero complesso inserito

SOTTRAZIONE COMPLESSA: CSUB

Obiettivo

Trovare la differenza fra due numeri complessi.

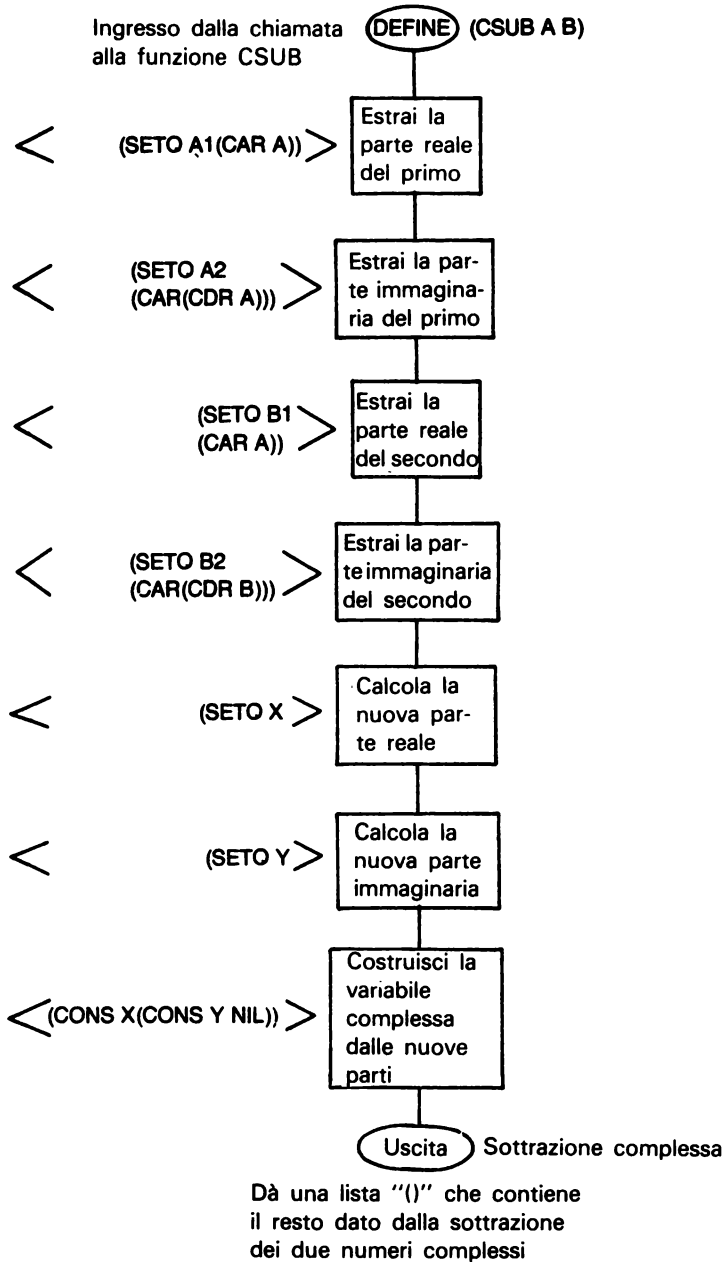
Programma

```
(DEFINE
(CSUB A B)
(SETQ A1
(CAR A))
(SETQ B1
(CAR B))
(SETQ B2
(CAR
(CDR B)))
(SETQ A2
(CAR
(CDR A)))
(SETQ X
(DIFFERENCE A1 B1))
(SETQ Y
(DIFFERENCE A2 B2))
(CONS X (CONS Y NIL))
)
```

Esecuzione campione

```
   (CSUB '(1 0)'(1 0))
   (0 0)

********************************

   (CSUB '(2 1)'(1 0))
   (1 1)

********************************

   END
```

Diagramma di flusso: sottrazione complessa

Dà una lista "()" che contiene
il resto dato dalla sottrazione
dei due numeri complessi

QUADRATO COMPLESSO: CSR

Obiettivo

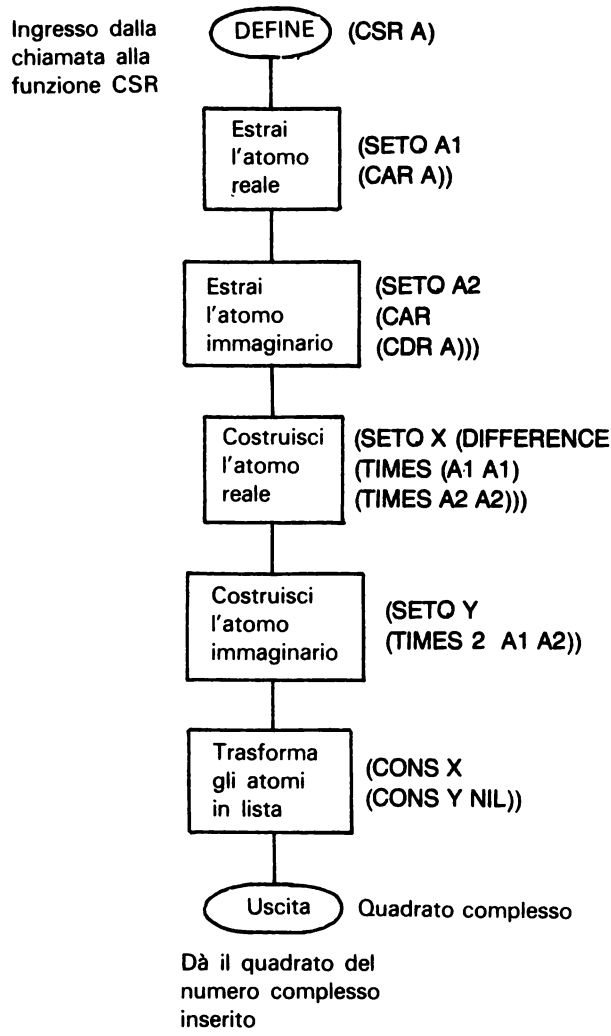
Trovare il quadrato di un numero complesso.

Programma

```
(DEFINE
(CSR A)
(SETQ A1
(CAR A))
(SETQ A2
(CAR
(CDR A)))
(SETQ X
(DIFFERENCE
(TIMES A1 A1)
(TIMES A2 A2)))
(SETQ Y
(TIMES 2 A1 A2))
(CONS X
(CONS Y NIL))
)
```

Esecuzione campione

```
   (CSR '(1 0))
   (1 0)

********************************

   (CSR '(1 2))
   (-3 4)

********************************

   END
```

Diagramma di flusso: quadrato complesso

RADICE QUADRATA COMPLESSA: CSQRT

Obiettivo

Trovare la radice quadrata di un numero complesso.

Programma

```
(DEFINE
(CSQRT A)
(SETQ A1 (CAR A))
(SETQ A2 (CAR (CDR A)))
(SETQ X
(EXPT
(QUOTIENT
(PLUS A1
(EXPT
(PLUS
(TIMES A1 A1)
(TIMES A2 A2))
.5))
2)
.5))
(SETQ Y (QUOTIENT A2
(TIMES 2
(EXPT
(QUOTIENT (EXPT
(PLUS
(TIMES A1 A1)
(TIMES A2 A2))
.5)
2)
.5))))
(CONS X (CONS Y NIL))
)
```

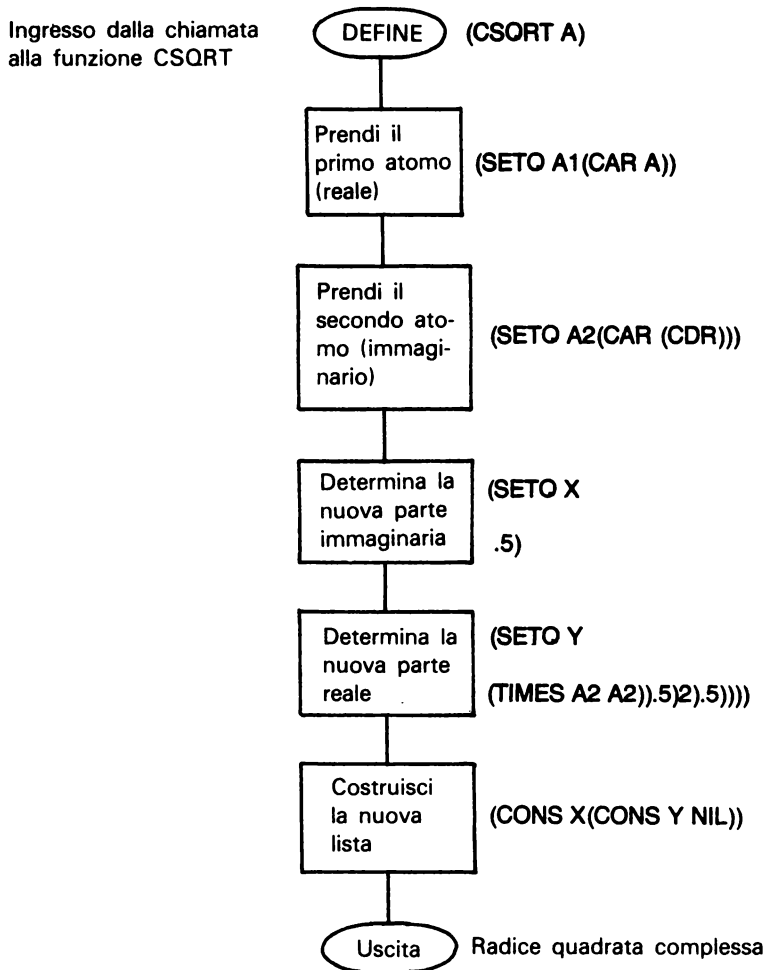
Esecuzione campione

```
   (CSQRT '(1 0))
   (1 0)

*******************************

   (CSQRT '(1 1))
   (1.09 .45)

*******************************

   (CSQRT '(1 2))
   (1.27 .78)

*******************************
```

```
(CSQRT '(3 4))
(2 1)
```

****

END

Diagramma di flusso: radice quadrata complessa

Dà una lista "()" che contiene la radice quadrata del numero complesso inserito

COSH(X)

Obiettivo

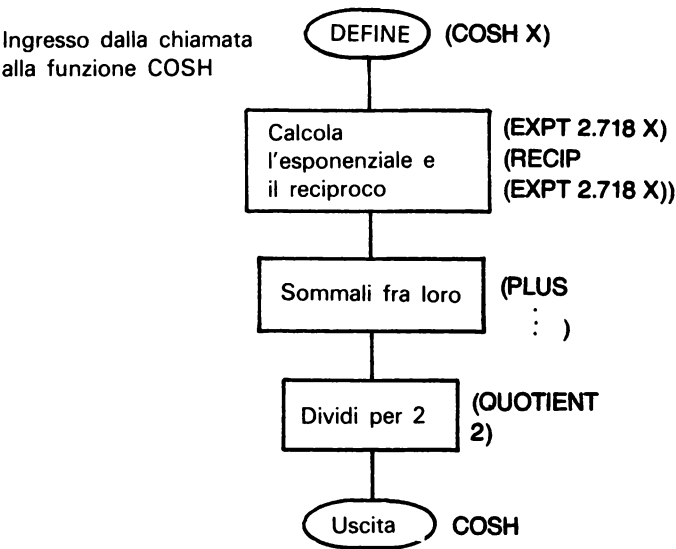
Trovare il coseno iperbolico di X.

Programma

```
(DEFINE
(COSH X)
(QUOTIENT
(PLUS
(EXPT 2.718 X)
(RECIP
(EXPT 2.718 X)))
2)
)
```

Esecuzione campione

```
   (SETQ X 1)
   (COSH X)
   (1.543

******************************

   (SETQ X 2)
   (COSH X)
   3.762

******************************

   (SETQ X 3)
   (COSH X)
   10.07

******************************

   END
```

Diagramma di flusso: COSH(X)

COTH(X)

Obiettivo

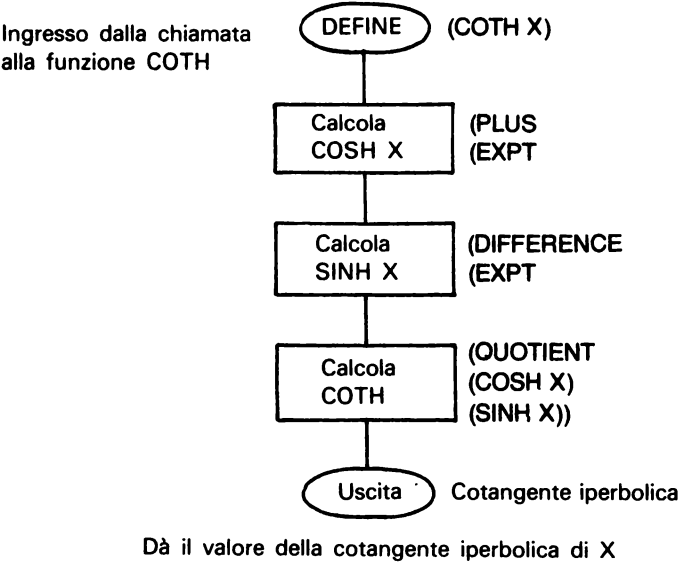
Trovare il valore della cotangente iperbolica di X.

Programma

```
(DEFINE
(COTH X)
(QUOTIENT
(PLUS
(EXPT 2.718
(MINUS X)))
(DIFFERENCE
(EXPT 2.718 X)
(RECIP
(EXPT 2.!718 X))))
)
```

Esecuzione campione

```
   (COTH 1)
   1.313

********************************

   (COTH 2)
   1.037

********************************

   (COTH 5)
   1.000

********************************

   END
```

Diagramma di flusso: COTH (X)

CONTEGGIO DEGLI ATOMI: CTA

Obiettivo

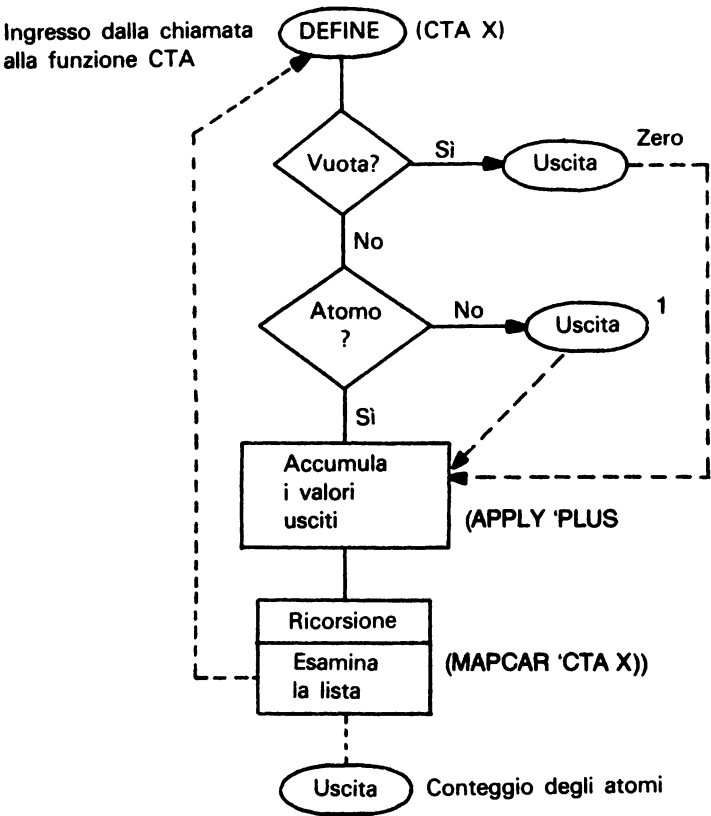
Contare il numero degli atomi nella S-espressione X.

Programma

```
(DEFINE (CTA X)
(COND
((NULL X)
0)
((ATOM X)
1)
(T
(APPLY 'PLUS
(MAPCAR 'CTA X)))))
```

Esecuzione campione

```
   (SETQ X (QUOTE (GIORGIO STA BENE)))
   (CTA X)
   3

*******************************

   (SETQ X '(MINUTI IN DUE ORE))
   (CTA X)
   4

*******************************

   (SETQ X '(COME TI SENTI OGGI LUCA))
   (CTA X)
   5

*******************************

   (SETQ X (QUOTE TOPI))
   (CTA X)
   1

*******************************

   END
```

Diagramma di flusso: conteggio degli atomi

Nota: le linee tratteggiate si riferiscono ai percorsi ricorsivi.

CSCH(X)

Obiettivo

Trovare il valore della cosecante iperbolica di X.

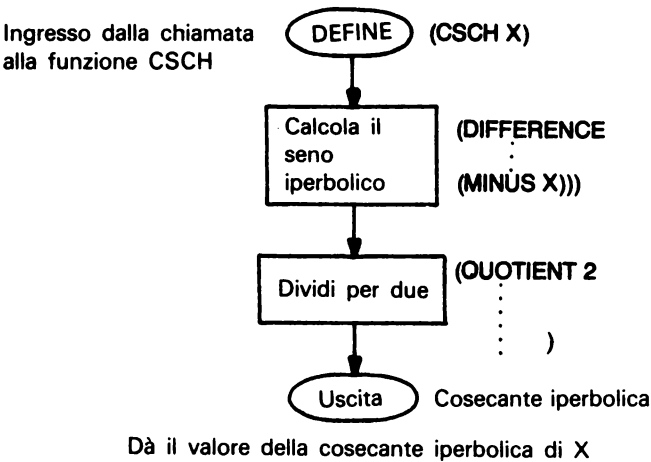
Programma

```
(DEFINE
(CSCH X)
(QUOTIENT 2
(DIFFERENCE
(EXPT 2.718 X)
ESPT 2.718
(MINUS X))))
)
```

Esecuzione campione

```
   (SETQ X .1
   (CSCH X)
   9.983

*******************************

   (SETQ X .2)
   (CSCH X)
   4.966

*******************************

   (CSCH 0.3)
   3.282

*******************************

   (CSCH 0.8)
   1.125

*******************************

   (CSCH 0.7)
   1.318

*******************************

   END
```

Diagramma di flusso: CSCH (X)

Dà il valore della cosecante iperbolica di X

PROFONDITÀ DELLE PARENTESI: DEPTH

Obiettivo

Trovare la profondità di una S-espressione, senza contare la prima e l'ultima parentesi. La profondità DEPTH è la misura del livello della sezione più interna di una struttura a parentesi annidate in una lista.

Programma

```
(DEFINE
(DEPTH L)
((NULL L)
0)
((ATOM (CAR L))
(DEPTH
(CDR L)))
(GREATERP (ADD1
(DEPTH (CAR L)))
(DEPTH (CDR L)))
(AD1
(DEPTH (CAR L))))
(T
(DEPTH (CDR L)))
))
```

Esecuzione campione

```
   (DEPTH '((D)((A(D)L))))
   3

*******************************

   (DEPTH '((((((K))))) J((T))))
   5

*******************************

   END
```

DERIVATA DI ACOS(X): DACOS

Obiettivo

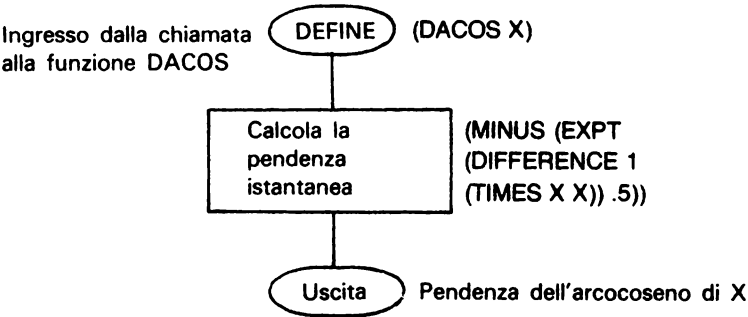
Trovare la derivata dell'arcocoseno di X.

Programma

```
(DEFINE
(DACOS X)
(MINUS
(EXPT
(DIFFERENCE 1
(TIMES X X))
.5))
)
```

Esecuzione campione

```
(DACOS 0.1)
-.99499

*******************************

(DACOS 0.8)
-.60000

*******************************

(DACOS 1.0)
-0

*******************************

(DACOS 0.15)
-.9887

*******************************

(DACOS 0.6)
-.8000

*******************************

(DACOS .95)
-.0312

*******************************

END
```

Diagramma di flusso: derivata di ACOS (X)

DERIVATA DI ACOT(X): DACOT

Obiettivo

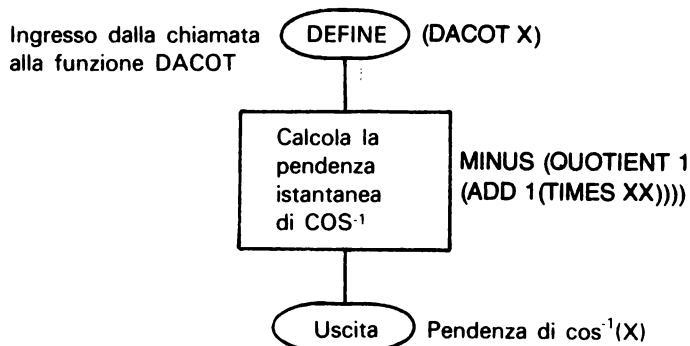
Trovare la derivata dell'arcocotangente di X.

Programma

```
(DEFINE
(DACOT X)
(MINUS
(QUOTIENT 1
(ADD1
(TIMES X X))))
)
```

Esecuzione campione

```
(SETQ X 0.1)
(DACOT X)
-.9901

********************************

(SETQ X 0.8)
(DACOT X)
-.6098

********************************

END
```

Diagramma di flusso: derivata di ACOT (X)

DERIVATA DI ASECH(X): DASECH

Obiettivo

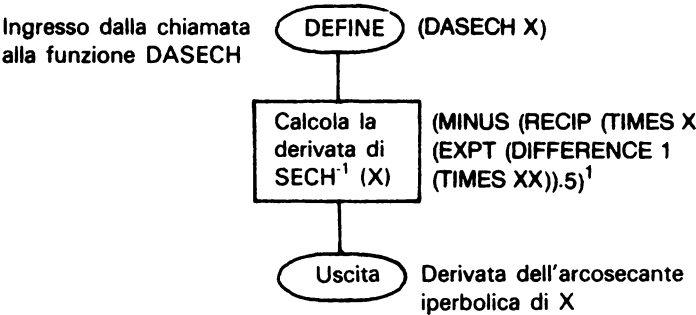
Trovare la derivata dell'arcosecante iperbolica di X.

Programma

```
(DEFINE
(DASECH X)
(MINUS
(RECIP
(TIMES X
EXPT
(DIFFERENCE 1
(TIMES X X))
.5))))
)
```

Esecuzione campione

```
   (DASECH .1)
   10.05

*******************************

   (DASECH .4)
   2.728

*******************************

   (DASECH .7)
   2.000

*******************************

   (DASECH .6)
   2.083

*******************************

   END
```

Diagramma di flusso: derivata di ASECH (X)

DERIVATA DI ATAN(X): DATAN

Obiettivo

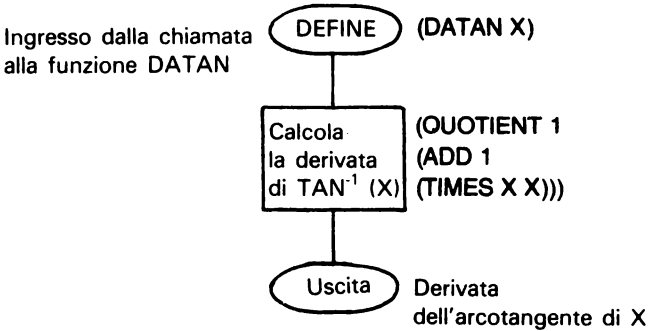
Trovare la derivata dell'arcotangente di X.

Programma

```
(DEFINE
(DATAN X)
(QUOTIENT 1)
(ADD1
(TIMES X X)))
)
```

Esecuzione campione

```
   (DATAN 0.1)
   0.9901

*******************************

   (DATAN 0.8)
   0.6098

*******************************

   (DATAN 0)
   1.000

*******************************

   (DATAN 0.2)
   0.9615

*******************************

   (SETQ X 0.5)
   (DATAN X)
   0.8000

*******************************

   END
```

Diagramma di flusso: derivata di ATAN (X)

DERIVATA DI ATANH(X): DATAH

Obiettivo

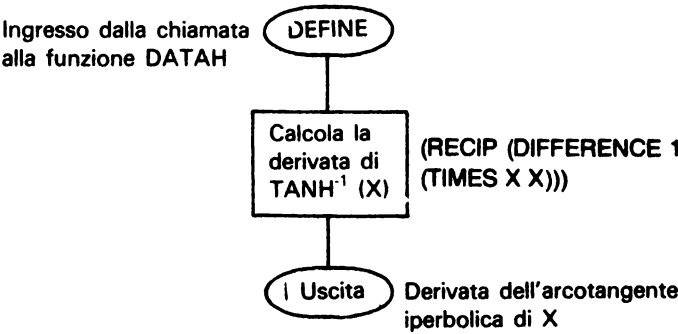
Trovare la derivata dell'arcotangente iperbolica di X.

Programma

```
(DEFINE
(DATAH X)
(RECIP
(DIFFERENCE 1
(TIMES X X)))
)
```

Esecuzione campione

```
   (DATAH 0.9)
   5.263

*******************************

   (DATAH 0.8)
   2.778

*******************************

   (DATAH 0.6)
   1.563

*******************************

   (DATAH 0.3)
   1.099

*******************************

   (DATAH 0.4)
   1.190

*******************************

   END
```

Diagramma di flusso: derivata di ATANH (X)

VISUALIZZARE L'ENNESIMO ATOMO: DIS N

Obiettivo

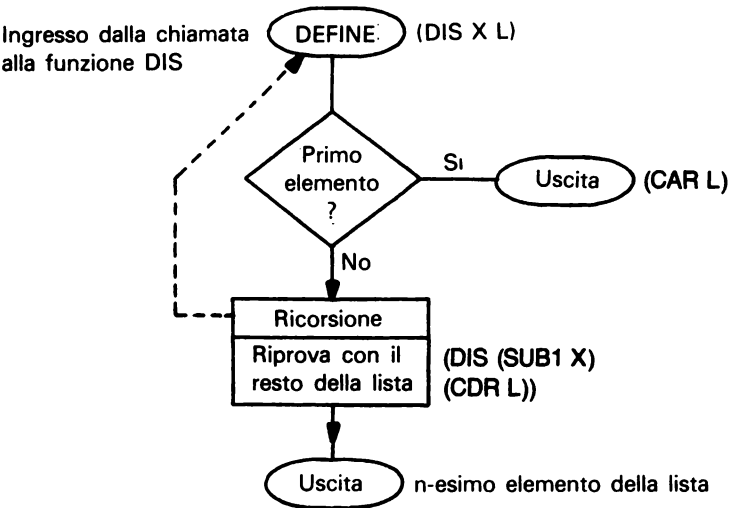
Visualizzare quale sia l'ennesimo atomo di una lista, senza modificare la lista stessa.

Programma

```
(DEFINE
(DIS X L)
(COND
((ZEROP
(SUB1 X))
(CAR L))
(T
(DIS (SUB1 X)
(CDR L)))
)))
```

Esecuzione campione

```
   (DIS 4 '(COME STA LUCA OGGI))
   OGGI

*******************************

   (DIS 3 '(ROSSO VERDE ROSA ARANCIONE BLU VIOLA))
   ROSA

*******************************

   (DIS 5 '(A B C D E F G H I J))
   E

*******************************

   (DIS 2 '(GIORGIO STA BENE))
   STA

*******************************

   (DIS 4 '(IN QUESTO LIBRO SI PARLA DEL LISP))
   SI

*******************************

   END
```

Diagramma di flusso: visualizzare l'ennesimo atomo

Nota: la linea tratteggiata si riferisce al percorso ricorsivo.

FATTORIALE

Obiettivo

Generare il fattoriale di X.

Programma

```
(DEFINE
(FATTORIALE X)
(PROG (F)
(SETQ F 1)
ANCORA
(COND
((ZEROP X)
(RETURN F)))
(SETQ F(TIMES F X))
(SETQ X (SUB1 X))
(GO ANCORA)))
```

Esecuzione campione

```
   (SETQ X 5)
   RUN
   120

*******************************

   (SETQ X 7)
   RUN
   5040

*******************************

   (SETQ X 9)
   RUN
   362880

*******************************

   (SETQ X 3)
   RUN
   6

*******************************

   (SETQ X 8)
   RUN
   40320

*******************************

   END
```

OGGETTO IN CADUTA: CADUTA

Obiettivo

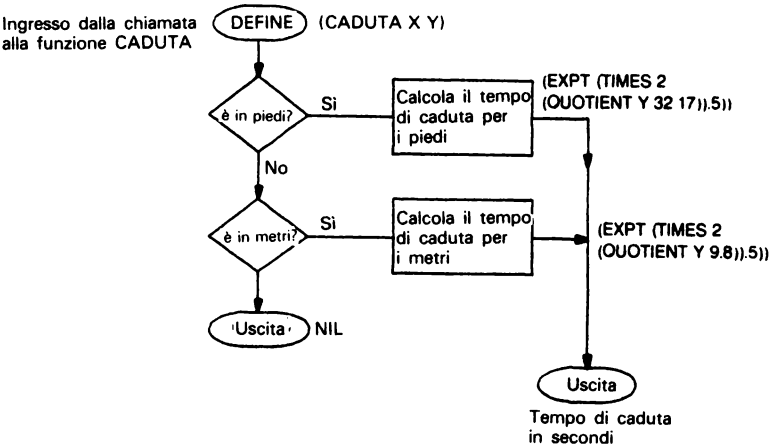
Trovare in quanti secondi un oggetto toccherà terra, se lasciato cadere da una determinata altezza, misura in piedi (F) o in metri (M).

Programma

```
(DEFINE
(CADUTA X Y)
(COND
((EQ X'F)
(EXPT
(TIMES 2
(QUOTIENT Y 32.17))
.5))
(EQ X'M)
(EXPT
(TIMES 2
(QUOTIENT Y 9.8))
.5))
))
```

Esecuzione campione

```
   (CADUTA 'F 1)
   0.2493

*******************************

   (CADUTA 'M 1)
   0.4518

*******************************

   (CADUTA 'F 49)
   1.745

*******************************

   END
```

Diagramma di flusso: oggetto in caduta

RECUPERO: REC

Obiettivo

Recuperare informazioni da una base di dati rappresentata dalla lista DATA. L'informazione da recuperare è chiamata I. Questa routine recupera solo la prima occorrenza dell'elemento cercato.

Programma

```
(DEFINE
(REC 1)
(PROG (INFO)
(SETQ INFO DATA)
ANCORA
(COND
((NULL INFO)
(RETURN NIL))
((MATCH I (CAR INFO))
(RETURN (CAR INFO))))
(SETQ INFO (CDR INFO))
(GO ANCORA)))
```

Esecuzione campione

Assumendo che la base di dati sia (LEONI TOPI GATTI CASA CANE),

```
    (SETQ DATA '(LEONI TOPI GATTO CASA CANE))

    (REC 'TOPI)
    TOPI

*******************************

    (REC 'TOPO)
    NIL

*******************************

    (REC 'CANE)
    CANE

*******************************
```

Se le informazioni nella base di dati DATA sono fra parentesi, ed entro ogni coppia di parentesi si trovano più atomi, si può usare il metodo che segue.
Supponiamo che la base di dati sia ((ALBERO TOPO)(ALBERO CASA)(GATTI E CANI)):

```
(SETQ DATA '((ALBERO TOPO)(ALBERO CASA)(CANI E GATTI))

(REC '(ALBERO CASA))
(ALBERO CASA)

*******************************

(REC '(GATTI))
NIL

*******************************

(REC & (GATTI))
(CANI E GATTI)

*******************************

END
```

INDUTTANZA: IND

Obiettivo

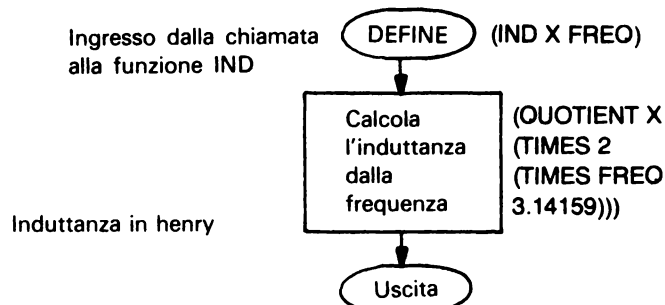
Trovare l'induttanza di un induttore.

Programma

```
(DEFINE
(IND X FREQ)
(QUOTIENT X
(TIMES 2
(TIMES FREQ
3.14159)))
)
```

Esecuzione campione

```
(SETQ X 1)
(SETQ FREQ 1000)
(IND X FREQ)
0.000159

*******************************

(SETQ X 2)
(SETQ FREQ 2000)
(IND X FREQ)
0.000159

*******************************

(IND 3 2000)
0.000318

*******************************

(IND 5 2000)
0.000397

*******************************

END
```

Diagramma di flusso: induttanza

PARTE IMMAGINARIA DI UN NUMERO COMPLESSO: PIC

Obiettivo

Trovare il valore della parte immaginaria di una variabile complessa.

Programma

```
(DEFINE
(PIC A)
(CAR
(CDR A))
)
```

Esecuzione campione

```
   (SETQ A '(1 0))
   (PIC A)
   0

*******************************

   (SETQ A '(2 1))
   (PIC A)
   1

*******************************

   (PIC '(3 2))
   2

*******************************

   (PIC '(4 3))
   3

*******************************

   (PIC '(7 9))
   9

*******************************

   END
```

Diagramma di flusso: parte immaginaria di un numero complesso

INSERIMENTO DI UN ATOMO A DESTRA: INSERDES

Obiettivo

Inserire un atomo dato in una lista, alla destra di un atomo specificato.

Programma

```
(DEFINE
(INSERDES AO AN L)
(COND
((NULL L)
())
((EQ (CAR L)
AO)
(CONS AO
(CONS AN
(CDR L))))
(T
(CONS (CAR L)
INSTR AO AN (CDR L))))
)))
```

Esecuzione campione

```
   (SETQ L '(GIORGIO BENE))
   (SETQ AO 'GIORGIO)
   (SETQ AN 'STA)
   (INSERDES AO AN L)
   (GIORGIO STA BENE)

*******************************

   (SETQ L '(IL DIAVOLO QUI)
   (SETQ AO 'DIAVOLO)
   (SETQ AN 'ABITA)
   (INSERDES AO AN L)
   (IL DIAVOLO ABITA QUI)

*******************************

   END
```

INSERIMENTO DI UN ATOMO A SINISTRA: INSERSIN

Obiettivo

Inserire un atomo dato in una lista, alla sinistra di un atomo specificato.

Programma

```
(DEFINE
(INSERSIN OA NA L)
(COND
((NULL L
())
((EQ (CAR L)
OA)
(CONS NA
(CONS (CAR L)
(CDR L))))
(T
(CONS (CAR L)
(INSERSIN OA NA (CDR L))))
)))
```

Esecuzione campione

```
(SETQ OA 'STA)
(SETQ NA 'GIORGIO)
(SETQ L '(STA BENE))
(INSERSIN OA NA L)
(GIORGIO STA BENE)

********************************

(INSERSIN 'PROGRAMMA 'QUESTO '(PROGRAMMA IN LISP))
(QUESTO PROGRAMMA IN LISP)

********************************

END
```

LEGGE DI JOULE: CAL

Obiettivo

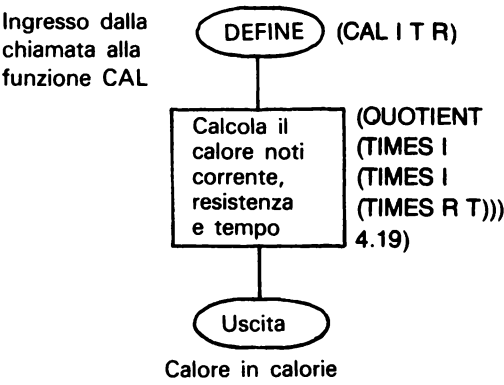
Trovare la quantità di calore (in calorie) dissipata da una corrente data, attraverso una resistenza data, in un tempo dato.

Programma

```
(DEFINE
(CAL I T R)
(QUOTIENT
(TIMES I
(TIMES I
(TIMES R T)))
4.19)
)
```

Esecuzione campione

```
   (SETQ I 1)
   (SETQ T 2)
   (SETQ R 3)
   (CAL I T R)
   1.432

********************************

   (CAL 4 5 6)
   114.558

********************************

   (CAL 7 8 9)
   842.005

********************************

   END
```

Diagramma di flusso: legge di Joule

LISTA DEI PRIMI ATOMI: LPA

Obiettivo

Produrre una lista i cui elementi siano i primi atomi di ciascuna delle sottoliste di una lista L data.

Programma

```
(DEFINE
(LPA L)
(COND
((NULL)
())
(T
(CONS (CAR (CAR L))
(LPA (CDR L))))
))
```

Esecuzione campione

```
   (LPA '((T R)(Y U)(I Z))
   (T Y I)

*******************************

   (SETQ L '((FG)(J K)(E R)))
   (LPA L)
   (F J E)

*******************************

   (SETQ L '((VGF TRD)(BHU KRF)(NKU DFT)))
   (LPA L)
   (VGF BHU NKU)

*******************************

   END
```

CORRISPONDENZA SELETTIVA: SEL

Obiettivo

Esaminare una lista che contiene valori numerici, trovare il più grande e quindi selezionare da una seconda lista l'elemento che si trova nella medesima posizione relativa.

Programma

```
(DEFINE((
(SEL (LAMBDA (L M)
(PROG (A B U V Y Z)
(SETQ A (CAR L))
(SETQ B (CAR M))
(SETQ U (CDR L))
(SETQ V (CDR M))
      ANCORA
(COND
((NULL U)
(RETURN B)))
(SETQ Y (CAR U))
(SETQ Z (CAR V))
(SETQ U (CDR U))
(SETQ V (CDR V))
(COND
((GREATER Y A)
(PROG2 (SETQ A Y)
(SETQ B Z))))
(GO ANCORA)
)))))
```

Esecuzione campione

```
(SETQ L '(1 8 56 7))
(SETQ M '(T O P I))
RUN
P

*******************************

END
```

VALORE MASSIMO DI UN VETTORE: VM

Obiettivo

Trovare il massimo fra i valori che costituiscono un vettore.

Programma

```
(DEFINE
(VM L)
(COND
((NULL (CDR L))
(CAR L))
((GREATERP (CAR L)
(CAR (CDR L)))
(VM
(CONS (CAR L)
(CDR (CDR L)))))
(T
(VM (CDR L)))
))
```

Esecuzione campione

```
   (VM '(4 8 6 9 3))
   9

*******************************

   (VM '(78 654 890 567))
   890

*******************************

   (VM '(11 111 8 1111))
   1111

*******************************

   END
```

RESISTENZE IN PARALLELO: RP

Obiettivo

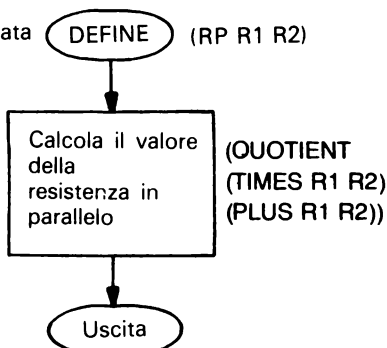
Trovare la resistenza totale di due resistori in parallelo.

Programma

```
(DEFINE
(RP R1 R2)
(QUOTIENT
(TIMES R1 R2)
(PLUS R1 R2))
)
```

Esecuzione campione

```
(SETQ R1 1)
(SETQ R2 1
(RP R1 R2)
0.5000

*******************************

(RP 1 2)
0.6000

*******************************

(RP 1 3)
0.7500

*******************************

(RP 2 8)
1.6000

*******************************

END
```

Diagramma di flusso: resistenze in parallelo

Resistenza dei resistori in parallelo, in ohm

VARIAZIONE PERCENTUALE: VP

Obiettivo

Trovare la variazione percentuale da Y a X.

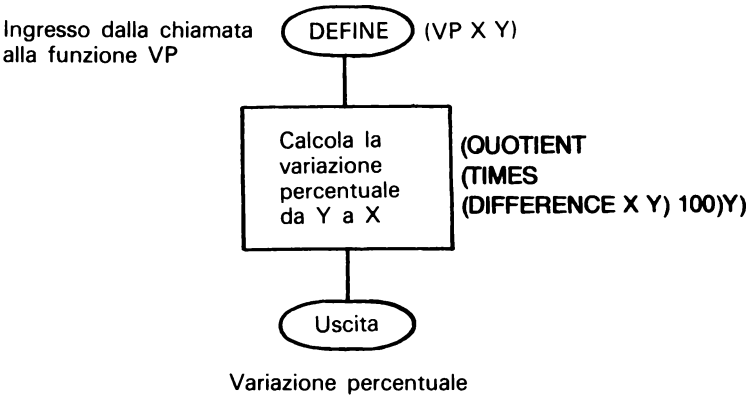
Programma

```
(DEFINE
(VP X Y)
(QUOTIENT
(TIMES
(DIFFERENCE X Y)
100)
Y)
)
```

Esecuzione campione

```
(VP 1 1)
0

******************************

(VP 2 1)
100.00

******************************

(VP 3 1)
200.00

******************************

(VP 4 1)
300.00

******************************

(VP 5 2)
150.00

******************************

END
```

Diagramma di flusso: variazione percentuale

Variazione percentuale

POTENZA: POT

Obiettivo

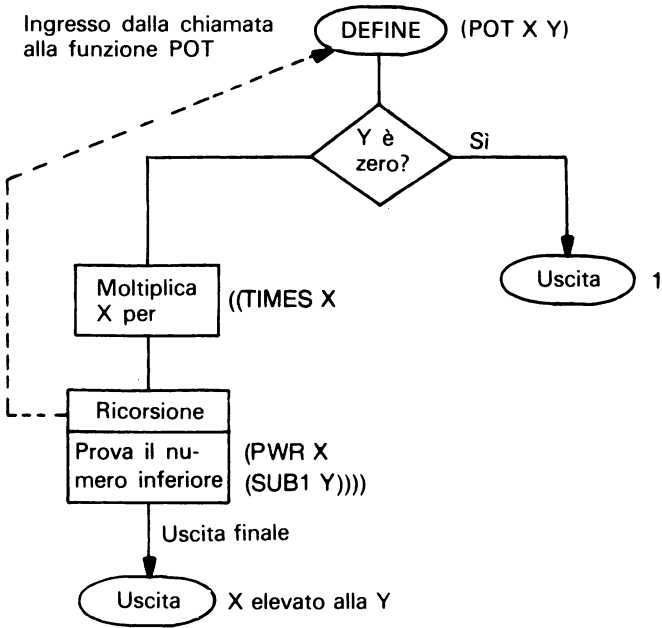
Elevare X alla potenza Y, dove Y può essere un intero positivo qualunque. Questa è una funzione ricorsiva.

Programma

```
(DEFINE (POT X Y)
(COND (ZEROP Y)
1)
((TIMES X
(POT X
(SUB 1 Y)))))
```

Esecuzione campione

```
(POT 4 5)
256

*******************************

(POT 6.9 0)
1

*******************************

(POT 54 1)
54

*******************************

(POT 3 8)
(6561)

*******************************

(POT 10 4)
(1000)

*******************************

(POT 5.5 3)
166.375

*******************************

(POT 4 4)
256

*******************************

END
```

Diagramma di flusso: potenza (per qualunque intero positivo)

Nota: la linea tratteggiata indica il percorso ricorsivo.

POTENZA (PRIMITIVA): POTPR

Obiettivo

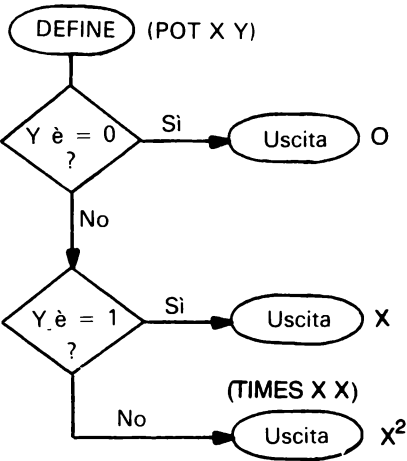
Elevare X alla potenza Y, dove Y può essere 0, 1 o 2. Questa è una funzione potenza primitiva.

Programma

```
(DEFINE (POT X Y)
(COND ((EQUAL Y 0)
1)
((EQUAL Y 1)
X)
((EQUAL Y 2)
(TIMES X X))))
```

Esecuzione campione

```
   (POT 5 0)
   1

*****************************

   (POT 67 1)
   67

*****************************

   (POT 34 2)
   1156

*****************************

   (POT 4.16 2)
   17.3056

*****************************

   (POT 567 1)
   567

*****************************

   (POT 34.9 0)
   1

*****************************

   END
```

Diagramma di flusso: potenza (primitiva)

PARTE REALE DI UN NUMERO COMPLESSO: PRC

Obiettivo

Trovare la parte reale di un numero complesso dato o evidenziare il primo elemento di una lista.

Programma

```
(DEFINE
(PRC A)
(CAR A))
```

Esecuzione campione

```
   (PRC '(4 5))
   4

******************************

   (PRC '(56 78))
   56

******************************

   (PRC '(-8 -90))
   -8

******************************

   (PRC '(2 67))
   2

******************************

   (PRC '(MELA ARANCIA))
   MELA

******************************

   (PRC '(A 56))
   A

******************************

   END
```

CONVERSIONE DA REALE A COMPLESSO: CRC

Obiettivo

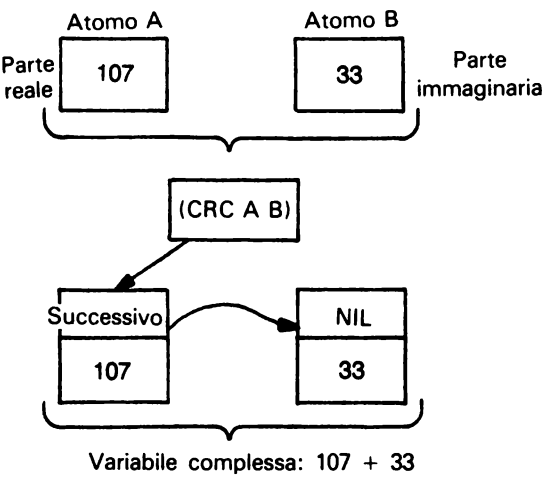
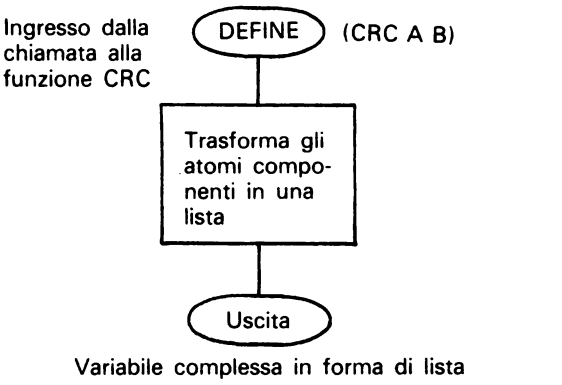
Prendere due numeri reali e trasformarli in un numero complesso. Una routine utile anche per riunire due elementi in una lista.

Programma

```
(DEFINE
(CRC A B)
(CONS A
(CONS B NIL))
)
```

Esecuzione campione

```
   (CRC 1 0)
   (1 0

*****************************

   (SETQ A 2)
   (SETQ B 3)
   (CRC A B)
   (2 3)

*****************************

   (SETQ A 4)
   (SETQ B 5)
   (CRC A B)
   (4 5)

*****************************

   (CRC 6 7)
   (6 7)

*****************************

   END
```

Diagramma di flusso: conversione da reale a complesso

MOMENTO POLARE DI UN RETTANGOLO: MPR

Obiettivo

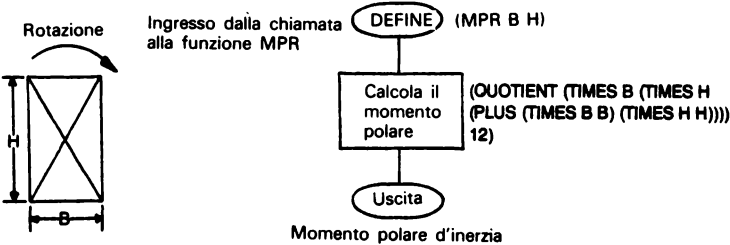
Trovare il momento polare d'inerzia di un rettangolo.

Programma

```
(DEFINE
(MPR B H)
(QUOTIENT
(TIMES B
(TIMES H
(PLUS
(TIMES B B
(TIMES H H))))
12)
)
```

Esecuzione campione

```
(SETQ B 1)
(SETQ H 1)
(MPR 1 2)
0.16667

******************************

(MPR 1 2)
0.83333

******************************

(MPR 1 3)
2.5000

******************************

END
```

Diagramma di flusso: momento polare di un rettangolo

ELIMINARE: ELIMIN

Obiettivo

Usare una lista come base di dati ed eliminare da essa delle informazioni. La base di dati è chiamata DATA, I è l'informazione da eliminare. Se l'informazione I era effettivamente presente in DATA viene stampata; in caso contrario, non c'è risultato.

Programma

```
(DEFINE
(ELIMIN I)
(COND ((MEMBER
I DATA)
(SETQ DATA
(DELETE I DATA)))
```

Esecuzione campione

Assumiamo che la base di dati contenga TOPO LEONE GATTO: allora, se I è LEONE

```
    (SETQ DATA '(TOPO LEONE GATTO))
    (SETQ I 'LEONE)
    (ELIMIN I)
    LEONE

******************************

    (SETQ I 'TOPI)
    (ELIMIN I)

******************************

    (SETQ I 'GATTO)
    (ELIMIN I)
    GATTO

******************************

    END
```

ELIMINARE UN ATOMO DA UNA LISTA: ELIMAT

Obiettivo

Eliminare la prima occorrenza di un atomo da una lista.

Programma

```
(DEFINE
(ELIM A L)
(COND
((NULL L)())
((EQ (CAR L) A)
(CDR L))
(T
(CONS (CAR L)
(ELIM A (CDR L))))
))
```

Esecuzione campione

```
    (ELIM 'SONO '(I TOPI SONO SIMPATICI))
    (I TOPI SIMPATICI)

*****************************

    (ELIM 'S '(T R S U))
    (T R U)

*****************************

    (ELIM 'RAGAZZE '(I RAGAZZI AMANO LE RAGAZZE))
    (I RAGAZZI AMANO LE)

*****************************

    (ELIM 'IO '(IO AMO LE BANANE))
    (AMO LE BANANE)

*****************************

    (SETQ A 'KEN)
    (SETQ L '(KEN AUTORE DEL LIBRO)
    (ELIM A L)
    (AUTORE DEL LIBRO)

*****************************

    END
```

ELIMINARE I NUMERI: ELIMNUM

Obiettivo

Eliminare da una lista selettivamente solo gli elementi che sono numeri, e lasciare gli altri atomi.

Programma

```
(DEFINE
(ELIMNUM L)
(COND
((NULL L
())
(NUMBERP (CAR L))
(ELIMNUM (CDR L)))
(T
(CONS (CAR L)
(ELIMNUM (CDR L))))
))
```

Esecuzione campione

```
   (ELIMNUM '(6 ORSI 7 TOPI 9 SCIMMIE))
   (ORSI TOPI SCIMMIE)

******************************

   (ELIMNUM '(3 TOPI SONO 2 VOLTE MEGLIO DI 4 GATTI))
   (TOPI SONO VOLTE MEGLIO DI GATTI)

******************************

   (ELIMNUM '(CI SONO 44 LEONI DI PIETRA))
   (CI SONO LEONI DI PIETRA)

******************************

   END
```

SOSTITUZIONE: SOST

Obiettivo

Sostituire la prima occorrenza di un atomo in una lista con un altro atomo.

Programma

```
(DEFINE
(SOST OA NA L)
(COND
((NULL L
())
((EQ (CAR L)
OA)
(CONS NA (CDR L)))
(T
(CONS (CAR L)
(SOST OA NA (CDR L))))
)))
```

Esecuzione campione

```
   (SETQ OA 'SONO)
   (SETQ NA 'SEMBRANO)
   (SETQ L '(I TOPI SONO SIMPATICI))
   (SOST OA NA L)
   (I TOPI SEMBRANO SIMPATICI)

******************************

   (SOST 'MELE 'BANANE '(LE MELE SONO FRUTTI))
   (LE BANANE SONO FRUTTI)

******************************

   END
```

INVERSIONE: INVER

Obiettivo

Invertire l'ordine degli atomi in una lista.

Programma

```
(DEFINE
(INV (LAMBDA (K) (PROG (J N)
(SETQ J K)
(SETQ N NIL)
    ANCORA
(COND
((NULL J)
(RETURN N)))
(SETQ N (CONS (CAR J) N))
(SETQ J (CDR J))
GO ANCORA)
)))))
```

Esecuzione campione

```
   (SETQ X '(A B C D))
   RUN
   (D C B A)

******************************

   (SETQ X '(1 6 8 4 8))
   RUN
   (8 4 8 6 1)

******************************

   (SETQ X '(T O P I))
   RUN
   (I P O T)

******************************

   (SETQ X '(L I S P))
   RUN
   (P S I L)

******************************

   END
```

Obiettivo

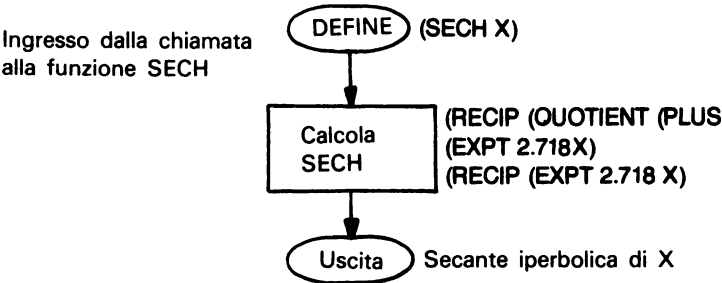
Trovare la secante iperbolica di X.

Programma

```
(DEFINE
(SECH X)
RECIP
(QUOTIENT
(PLUS
(EXPT 2.718 X)
RECIP
(EXPT 2.718 X)))
2))
)
```

Esecuzione campione

```
(SECH .1)
.9950

******************************

(SECH .2)
.9803

******************************

(SECH .5)
.8868

******************************

(SETQ X .6)
(SECH X)
.8435

******************************

END
```

Diagramma di flusso: SECH (X)

Nota: l'atomo 2.718 rappresenta la costante matematica e.

SINH(X)

Obiettivo

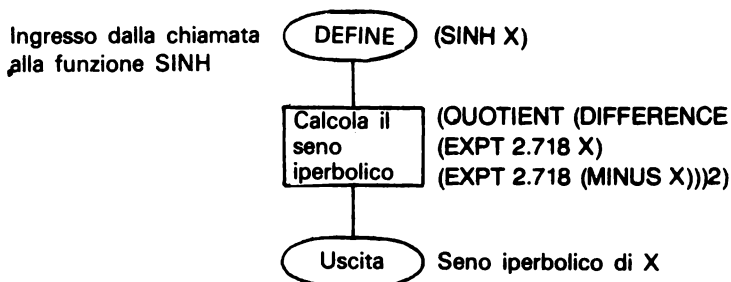
Trovare il valore del seno iperbolico di X.

Programma

```
(DEFINE
(SINH X)
(QUOTIENT
(DIFFERENCE
(EXPT 2.718 X)
(EXPT 2.718
(MINUS X)))
2)
)
```

Esecuzione campione

```
(SETQ X 1)
(SINH X)
1.175

******************************

(SETQ X 2)
(SINH X)
3.267

******************************

(SETQ X 3)
(SINH X)
10.02

******************************

END
```

Diagramma di flusso: SINH (X)

Obiettivo

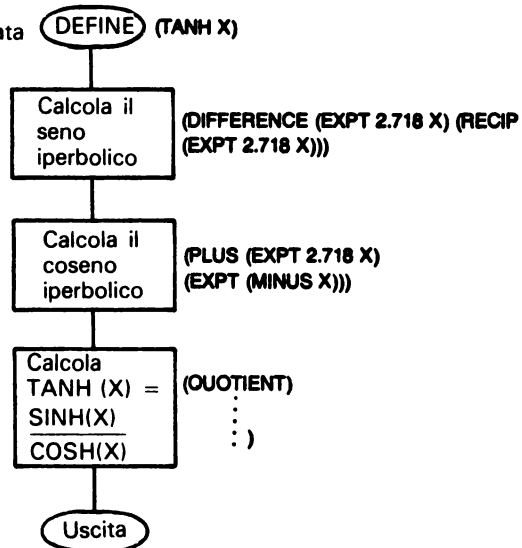
Trovare il valore della tangente iperbolica di X.

Programma

```
(DEFINE
(TANH X)
(QUOTIENT
(DIFFERENCE
(EXPT 2.718 X)
RECIP
(EXPT 2.718 X)))
(MINUS X))))
)
```

Esecuzione campione

```
(TANH 2)
.9640

******************************

(TANH 1)
.7616

******************************

(TANH 4)
.9993

******************************

END
```

Diagramma di flusso: TANH (X)

SOMMA VETTORIALE: VADD

Obiettivo

Effettuare la somma di due vettori qualunque.

Programma

```
(DEFINE
(VADD V1 V2)
(COND
((NULL V1)
V2)
((NULL V2)
V1)
(T
(CONS
(PLUS (CAR V1)
(CAR V2)
(VADD (CDR V1)(CDR V2))))
)))
```

Esecuzione campione

```
(SETQ V1 '(1 2 3 4))
(SETQ V2 '(5 6 7 8))
(VADD V1 V2)
(6 8 10 12)

******************************

(VADD '(6 7 8)'(3 3 3))
(9 10 11)

******************************

(VADD '(2 2 2 2) '(4 5 6 7))
(6 7 8 9)

******************************

END
```

PRODOTTO VETTORIALE: PV

Obiettivo

Trovare il prodotto vettoriale di due vettori qualunque.

Programma

```
(DEFINE
(PV A B)
(SETQ X1)
(CAR A))
(SETQ X2
(CAR B))
(SETQ Y1
(CAR (CDR A)))
(SETQ Y2
(CAR (CDR B))
(SETQ Z1
(CAR (CDR (CDR A))))
(SETQ Z2
(CAR (CDR (CDR B))))
(SETQ X3
(DIFFERENCE
(TIMES Y1 Z2)
(TIMES Z1 Y2)))
(SETQ Y3
(DIFFERENCE
(TIMES Z1 X2)
(TIMES X1 Z2)))
(SETQ Z3
(DIFFERENCE
(TIMES X1 Y2)
(TIMES X2 Y1)))
CONS X3
(CONS Y3
(CONS Z3 NIL)))
)
```

Esecuzione campione

```
(PV '(1 2 3)'(4 5 6))
(-3 6 -3)
```

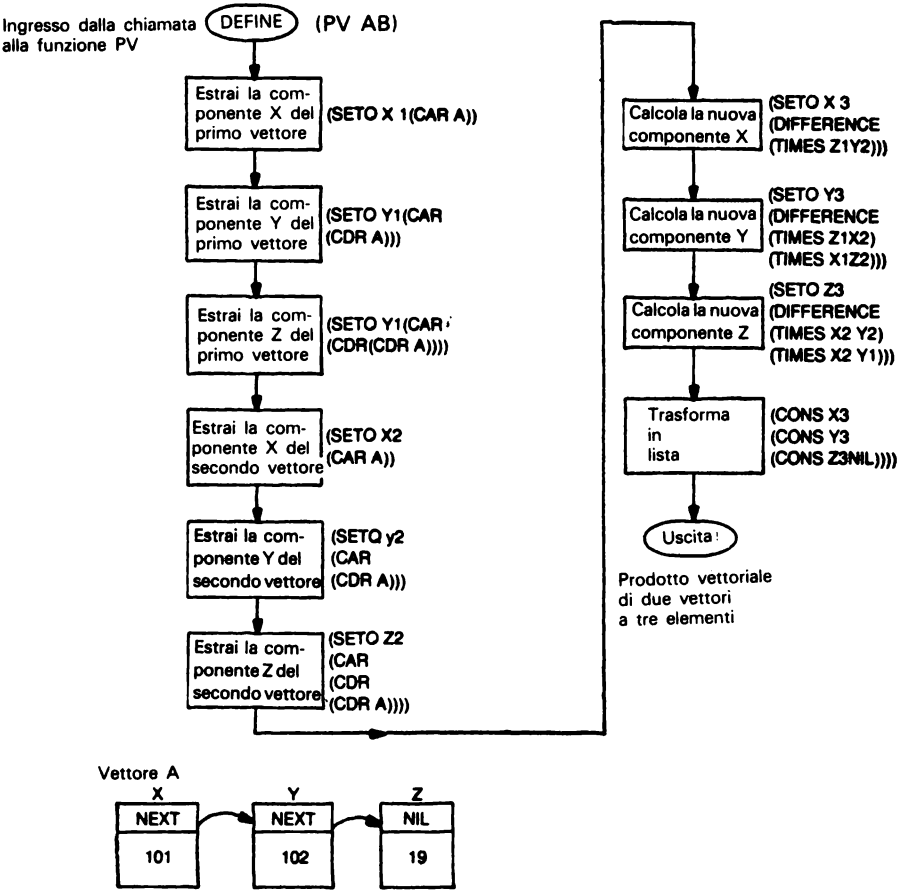
```
******************************
```

```
(PV '(7 8 9)(10 11 12))
(-3 6 -3)
```

```
******************************
```

```
(PV '(1 2 3)'(7 8 9))
(-6 12 -6)
```

```
******************************

   (PV '(1 2 3)'(12 11 10))
   (-13 26 -13)

******************************

   (PV '(3 1 2)'(4 5 6))
   (-4 -10 11)

******************************

   END
```

Diagramma di flusso: prodotto vettoriale

PRODOTTO DI UN VETTORE PER UNO SCALARE: PVS

Obiettivo

Moltiplicare un vettore per uno scalare e creare un nuovo vettore.

Programma

```
(DEFINE
(PVS X L)
(COND
((NULL L)
())
(T
(CONS
(TIMES X
(CAR L))
(PVS X
(CDR L))))
))
```

Esecuzione campione

```
   (PVS 5 '(3 5 6 7))
   (15 25 30 35)

******************************

   (PVS 8 '(6 5 4 6))
   (48 40 32 48)

******************************

   (PVS 10 '(21 17 13 4))
   (210 170 130 40)

******************************

   END
```

CONTROLLO DI VETTORE NULLO: NULVET

Obiettivo

Controllare un vettore, per stabilire se i suoi elementi sono tutti nulli.

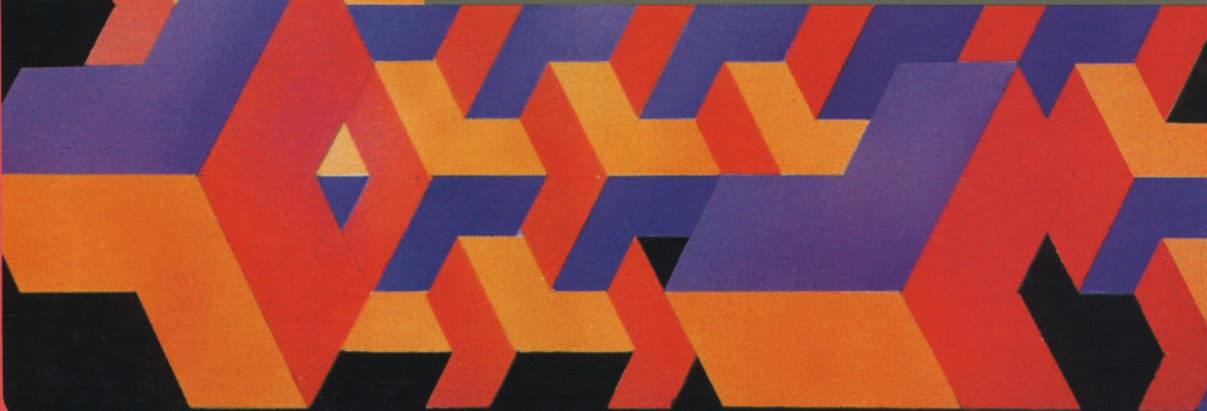
Programma

```
(DEFINE
(NULVET L)
(COND
((NULL L)
())
T)
((ZEROP
(CAR L))
(NULVET
(CDR L)))
(T NIL)
))
```

Esecuzione campione

```
   (NULVET '(0 0 9 0))
   NIL

*****************************

   (NULVET '(0 0 0 0 0 0))
   T

*****************************

   (NULVET '(8 0 7 0 6))
   NIL

*****************************

   (NULVET '(0 0))
   T

*****************************

   END
```

Finito di stampare nel settembre 1984 presso Lito Velox - Trento
Fotocomposto e impaginato da Lito Velox - Trento
Printed in Italy

Il Lisp è un linguaggio interpretato, come il Basic, ma è profondamente diverso da questo: è orientato all'elaborazione di liste e alla descrizione di processi, è un linguaggio "funzionale" (tutte le costruzioni sono funzioni), presenta forti somiglianze con i linguaggi della logica matematica, da cui è derivato; è chiaro e molto duttile, relativamente facile da comprendere. Tutto questo lo ha reso il linguaggio più diffuso negli studi sull'intelligenza artificiale.
Questo volume presenta gli elementi essenziali del Lisp in modo piano e accessibile, con uno stile reso ancora più leggibile dall'impostazione a "domanda e risposta" e riporta e commenta circa 60 fra programmi e routine di immediata applicazione.

franco muzzio editore

L. 19.000 ISBN 88-7021-256-4

28 Ken Tracton

INTRODUZIONE AL LISP